MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 1329 *Nov 1991*

The Supercomputer Toolkit: A general framework for special-purpose computing

Harold Abelson[1], Andrew A. Berlin[1], Jacob Katzenelson[2],
William H. McAllister[3], Guillermo J. Rozas[1],
Gerald Jay Sussman[1], and Jack Wisdom[4]

Abstract

The Toolkit is a family of hardware modules (processors, memory, interconnect, and input-output devices) and a collection of software modules (compilers, simulators, scientific libraries, and high-level front ends) from which high-performance special-purpose computers can be easily configured and programmed. The hardware modules are intended to be standard, reusable parts. These are combined by means of a user-reconfigurable, static interconnect technology. The Toolkit's software support, based on novel compilation techniques, produces extremely high-performance numerical code from high-level language input, and will eventually automatically configure hardware modules for particular applications.

We have completed fabrication of the Toolkit processor module, and an eight-processor configuration is running at MIT. As a demonstration of the power of the Toolkit approach, we have used the prototype Toolkit to perform an integration of the motion of the Solar System in a computation that extends previous results by nearly two orders of magnitude.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-3202 and by the National Science Foundation under grant number MIP-9001651. Andrew Berlin's was partially supported by an IBM fellowship.

[1]Department of Electrical Engineering and Computer Science, MIT.
[2]Department of Electrical Engineering, Technion—Israel Institute of Technology.
[3]Hewlett–Packard Corp.
[4]Department of Earth, Atmospheric, and Planetary Sciences, MIT.

The Supercomputer Toolkit: A general framework for special-purpose computing[1]

Special-purpose computational instruments will play an increasing role in the practice of science and engineering. Although general-purpose supercomputers are becoming more available, there are significant applications for which it is appropriate to construct special-purpose dedicated computing engines. The Supercomputer Toolkit is intended to make the construction and programming of such special-purpose computers routine and inexpensive, in some cases even automatic.

The Toolkit is a family of hardware modules (processors, memory, interconnect, and input-output devices) and a collection of software modules (compilers, simulators, scientific libraries, and high-level front ends) from which high-performance special-purpose computers can be easily configured and programmed. The hardware modules are intended to be standard, reusable parts. These are combined by means of a user-reconfigurable, static interconnect technology. The Toolkit's software support, based on novel compilation techniques, produces extremely high-performance numerical code from high-level language input, and will eventually automatically configure hardware modules for particular applications.

Our Supercomputer Toolkit is intended to help bring scientists and engineers back into the design loop for their computing instruments. Traditionally, they have been intimately involved in the development of their instruments. Computers, however, have been treated differently—scientists are primarily users of general-purpose computers supplied by a few remote vendors. Because a general-purpose computer is often not well-organized for a particular problem, the construction of appropriate software can be a long and complex task. By contrast, a computer whose design is specialized to a particular problem can be straightforward to program for the application for which it was designed. Moreover, the specialized computer can become an ordinary experimental instrument, or a component of a measurement tool or a process-control system, belonging to the group that made it.

Over the past three years, the MIT Project for Mathematics and Computation and Hewlett-Packard's Information Architecture Group have been collaborating on the design and construction of a prototype Supercomputer

[1] with apologies to Joel Moses [9].

Toolkit, and a system configured with eight processors is now running at MIT.

This prototype is targeted at numerical computations where performance is limited by the need to integrate systems of ordinary differential equations. Such computations are characterized by a bottleneck in scalar floating-point performance rather than in I/O or in memory bandwidth. These computations are typically not easy to vectorize. Highly pipelined vector processors do not do well on them because the state variables of the system must in general be updated by computing different expressions.

For suitable applications, we expect the Supercomputer Toolkit approach to provide substantial advantages over general-purpose supercomputers. This is not only because of the dedicated hardware, but because the Toolkit's novel compilation strategy generates outstandingly efficient code from general-purpose library modules and programs written in high-level languages. Benchmarks integrating systems of ordinary differential equations confirm that a single Toolkit processor, programmed in highly abstract Lisp code, achieves scalar floating-point performance equivalent to a Cray 1S programmed in Cray Fortran. While this does not match the speed of the fastest available supercomputer, the relative price advantage of the Toolkit allows it to be used for applications that would be otherwise infeasible. We have already used our eight-processor configuration to perform a computation of major scientific significance—an integration, running for 1000 hours, of the complete Solar System, which improves upon previous computations by almost two orders of magnitude.

In general, we envision that the Toolkit will be used as follows: One begins with an algorithm that performs the costly inner loop of a computation that warrants constructing a special-purpose machine. For example, the simulation part of a multidimensional optimization in the design of an analog circuit, or the integration of the differential equations required for real-time control of a nonlinear process, are appropriate for Toolkit implementation. The Toolkit software compiles the program, targeted for a number of different Toolkit hardware configurations, some proposed by the user, others generated automatically by the Toolkit compiler itself. The compiler also produces, for each configuration, a simulation that the user can run on the host machine to help evaluate price-performance tradeoffs. After a configuration has been selected, the user obtains the required modules, wires them together, and connects the machine to a host computer. The configuration is verified by

means of diagnostics that are automatically generated and loaded from the host. The target program is then loaded, and the new machine is ready to be used by host programs as a back-end processor.

The Toolkit's advantage arises from two architectural principles, coupled with supporting compiler technology. The first principle is the use of *problem-specific communication paths.* Starting with a particular algorithm, one can often find a static arrangement of interprocessor communication paths that admits nearly optimal utilization of processing power for that algorithm. Machines with specialized communication paths are currently built to implement extremely high-performance signal-processing systems, but these machines are expensive, because each one is manually configured and programmed. We believe that for many important scientific applications, high-performance configurations can be generated in a straightforward way, perhaps even automatically, and easily constructed and programmed using Toolkit modules.

The second principle is the use of *synchronous ultra-long instruction word* machines. Even in problems that are not vectorizable, scientific applications typically have substantial data-independent structure. One can configure a totally synchronous machine, in which most interprocessor communication is statically scheduled at compile time. In effect, the multiple VLIW execution units of the machine are programmed as a single ultra-long instruction word processor. This organization eliminates the need for synchronization, bus protocols, run-time handshaking, or any operating-system overhead.

The Toolkit's compiler uses a novel strategy based upon partial evaluation [3, 4]. It exploits the data-independence of typical numerical algorithms to generate exceptionally efficient object code, even from source programs that are expressed in terms of highly abstract components. This has enabled us to develop a library of symbolic manipulation components to support the automatic construction of simulation codes. As a measure of success, our Solar-system simulation code, constructed with this library, achieves 98% utilization of the Toolkit's available floating-point performance.

The Toolkit approach has obvious limitations. Neither our hardware architecture nor our interconnection technology can be expected to scale to systems with many hundreds of processors. On the other hand, the Toolkit does realize a means, practical within the limits of current technology, to provide relatively inexpensive supercomputer performance for an important class of problems.

Section 1 of this paper describes the Toolkit hardware. Section 2 presents the low-level programming model. Section 3 presents the high-level programming model, illustrated by showing how to integrate systems of ordinary differential equations starting from highly abstract programs. Section 4 describes our benchmark application to long-term integrations of the Solar System.

1 Toolkit Hardware

At the highest level, the Toolkit hardware is a network of processors, memories, and I/O modules. The topology of the network is chosen or suggested by the user, possibly with the help of the compiler. In our prototype, each Toolkit hardware module is an individual board. The Toolkit processor module is the only hardware module we have fabricated thus far, and is the only one discussed in this paper. We hope to design other hardware modules to provide a faster host interface, A/D and D/A channels, and a mass-memory module.

Each processor is by itself a high performance computer with its own private memory for code and data. Processors have two I/O ports that may be connected with fine-pitch ribbon cables to a number of other similar ports. Limiting the number of ports to two per processor simplifies the hardware design, yet permits a variety of network graphs. Any connection graph may be implemented by placing a processor on each branch of the graph, as shown in figure 1. With this method, the number of branches at any node is constrained by the bus fanout limits. The prototype has been tested with eight ports sharing a common bus.

In addition to the data communication paths there is a separate "global flag". This one-bit signal can be set and sensed by each processor. Like the data cables, the user manually configures this flag. Its primary use is for software synchronization as described below in section 2.6.

Once the customized Toolkit is assembled, it appears to the programmer that all elements run in lock step. The communication paths are under complete control of the software. There is no hardware for arbitration or bus protocol. The program specifies the source and sink(s) for data on each bus on each cycle. Software convention determines whether the bus value is data, address, or control information.

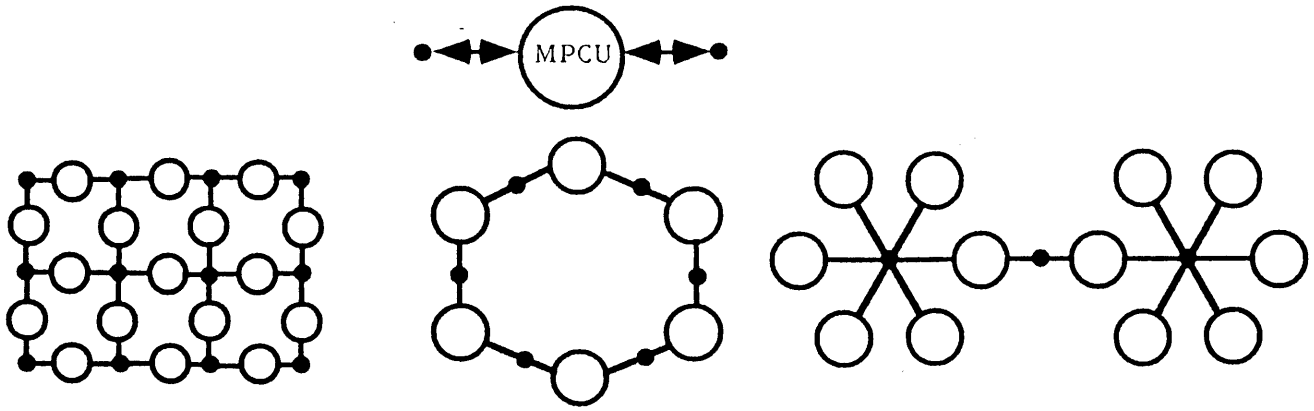


Figure 1: Each Toolkit processor module has two bidirectional I/O ports. Any interprocessor connection graph can be configured with each node having degree up to about 8. The figure shows how to build various network architectures: a mesh, a ring and communicating clusters.

The prototype Toolkit is housed in a minicomputer chassis borrowed from an existing Hewlett-Packard product line (HP9000/850). Our current collection of eight processor boards consumes about 1200 watts. All cables for data, clocks, global flag, and host communication connect to the front edge of each processor board. A user assembles a machine by plugging in the required modules and connecting the cables appropriately. When a particular machine is no longer needed, it can be disassembled, and its modules can be reassembled into other configurations.

1.1 Toolkit Processor overview

The processor design was guided largely by two factors: the class of problems we were targeting; and the use of off-the-shelf technology available in 1989. The class of problems is the solution of systems of ordinary differential equations—computations characterized by a large number of floating-point operations on a relatively small amount of data. Given the very small design team, the technology choices were confined to readily available commercial parts. No custom chips are used in the design, and all components use TTL

logic levels.

The processor architecture is centered around a high-performance floating-point chip set. The remaining hardware is designed to feed operands to the floating-point chips without interruption. Figure 2 shows a block diagram of the processor with the ALU, Multiplier, register file, data memories, address generators, I/O ports, instruction memory, and program sequencer. Not shown in the diagram are the host interface, clock generator, and some smaller buses.

The prototype Toolkit processor contains about 220 integrated circuits laid out on a 13 × 15 inch printed-circuit board as shown in figure 3. The eight-layer board has 8–10 mil traces with 8 mil spaces. Trace impedances range from 45 ohms to 80 ohms, with the higher impedance layers used for the critical I/O port wiring. The cycle time is 84 ns, which results in an instruction rate of 11.9 million instructions per second.

1.2 Floating-point unit

The heart of the processor is a pair of Bipolar Integrated Technology (B.I.T.) floating-point math chips [5]. The ALU chip (B2120A-25) does all arithmetic and logical operations in 25ns. The Multiplier chip (B2110A-55) performs double-precision (64-bit) multiplication in 55ns. Both chips have 32-bit I/O paths and full 64-bit internal paths. To simplify the architecture and timing, all operations are done in double precision. Operations complete in one cycle except for the Multiplier's divide and square root, which are multi-cycle operations.

Separate instruction bits to each math chip allow the ALU and Multiplier to be controlled independently. The peak floating-point performance is therefore two operations per cycle, or 23.8 MFLOPS per processor at the current 84ns clock rate. Section 2.7 below describes an inner-product routine that runs at this peak rate. In practice, we have been able to sustain roughly half this rate—about 12 MFLOPS per processor—on the applications of interest.

1.3 Buses and Register File

As illustrated in figure 2, the toolkit processor architecture contains a central five-port register file that communicates with each of the data memories, with the math chips, and with the I/O ports. The five ports of the register file are

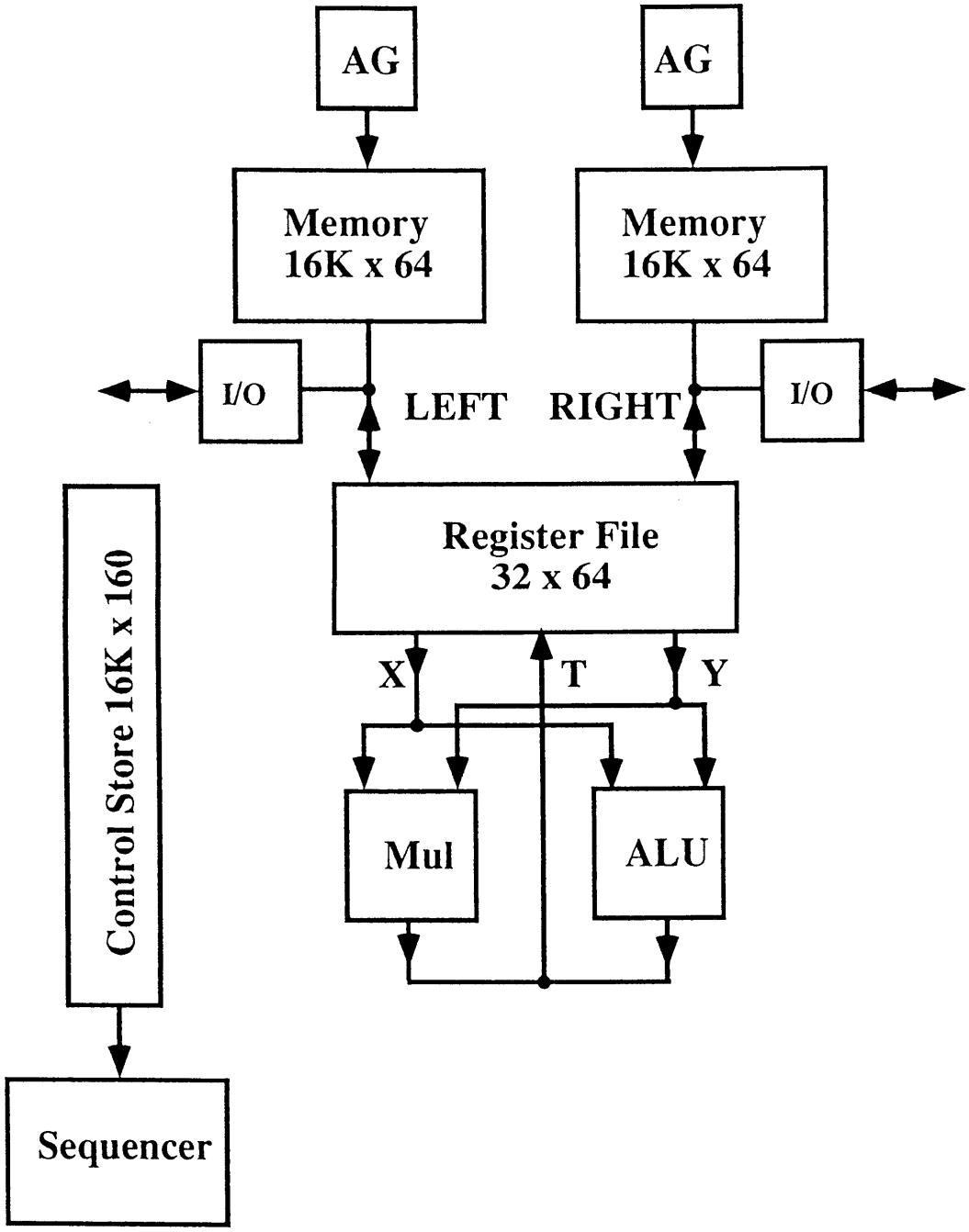


Figure 2: The Toolkit processor consists of floating-point unit (ALU and Multiplier), a register file, two data memories, each with its own address generator, two bidirectional I/O ports, instruction memory, and a program sequencer. The figure shows the major buses: X, Y, T, LEFT, and RIGHT. Not shown are the side paths, the internal paths in the Math chips, the communication between the sequencer and the rest of the machine, and the host interface.

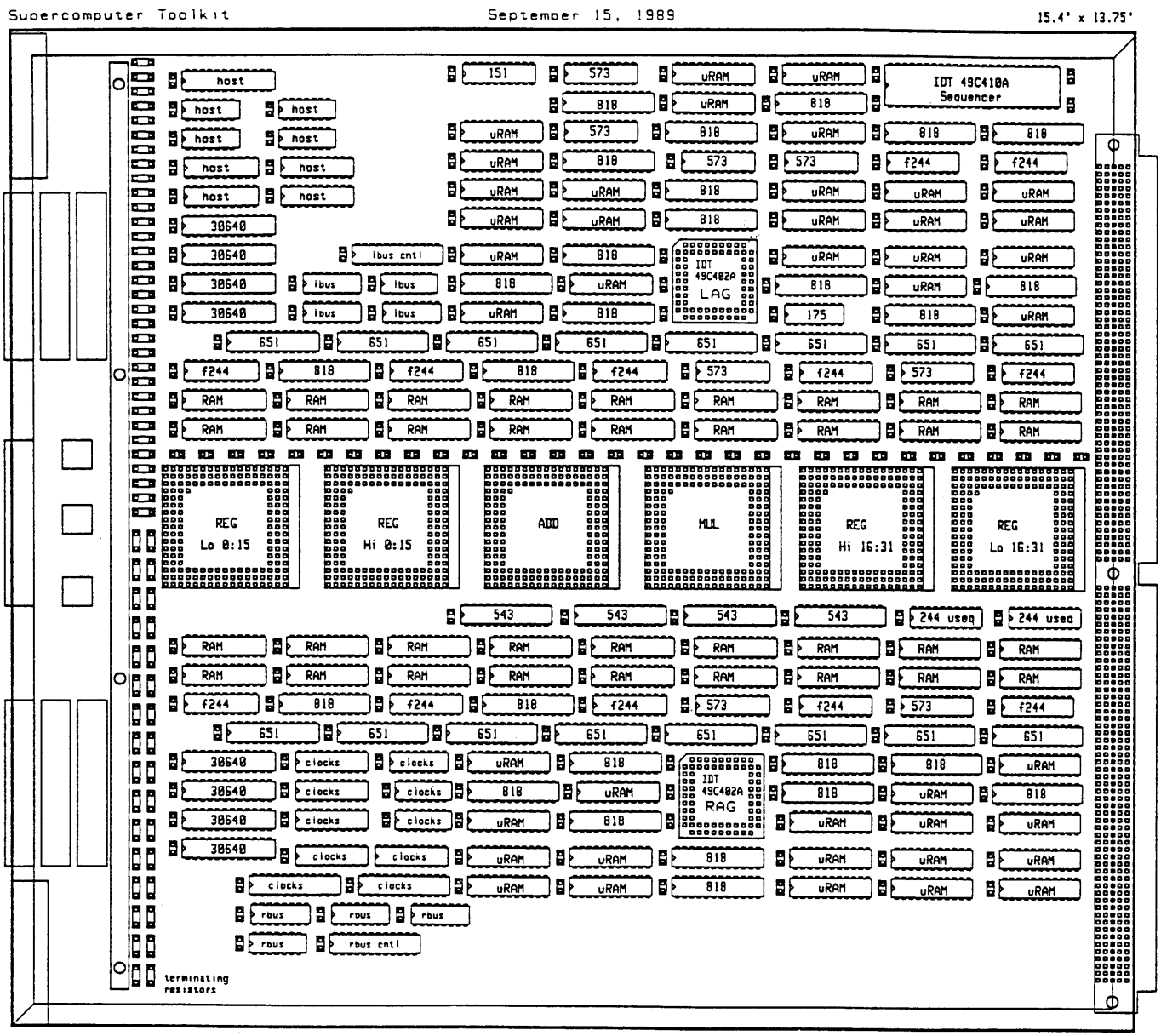


Figure 3: The Toolkit processor module is laid out on a 13" × 15" printed-circuit board. The six large pin grid arrays consist of the four register files, the ALU and the Multiplier. These chips occupy the center strip of the board with an almost symmetric complement of ICs surrounding them. The other specialized chips are the left and right address generators in the 68-pin square ICs, and the microprogram sequencer, the 48-pin dual inline package chip.

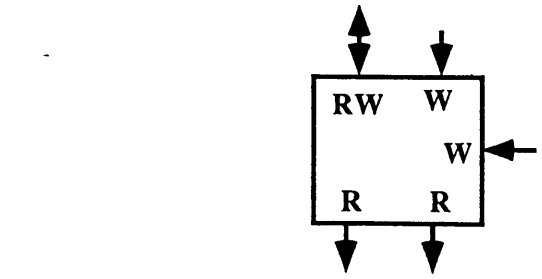
connected to five buses. The X and Y buses carry operands from the register file to the math chips, while the T bus returns results produced by the math chips to the register file. The LEFT bus connects the register file to the left memory and to the left I/O port, while the RIGHT bus connects the register file to the right memory and to the right I/O port. Even though each I/O port lies on the same bus as a memory, the processor timing does not permit the I/O port and memory to communicate directly with each other; instead, all communication is performed via the register file.

The central register file contains thirty-two 72-bit wide registers, to accommodate 64-bit IEEE double-precision floating-point numbers with eight bits of parity protection. The entire register file is constructed from four B.I.T. register-file chips (B2210A), each of which provides sixty-four 18-bit wide registers. These four chips are organized as two dual-B2210A register-files, which hold duplicate data.

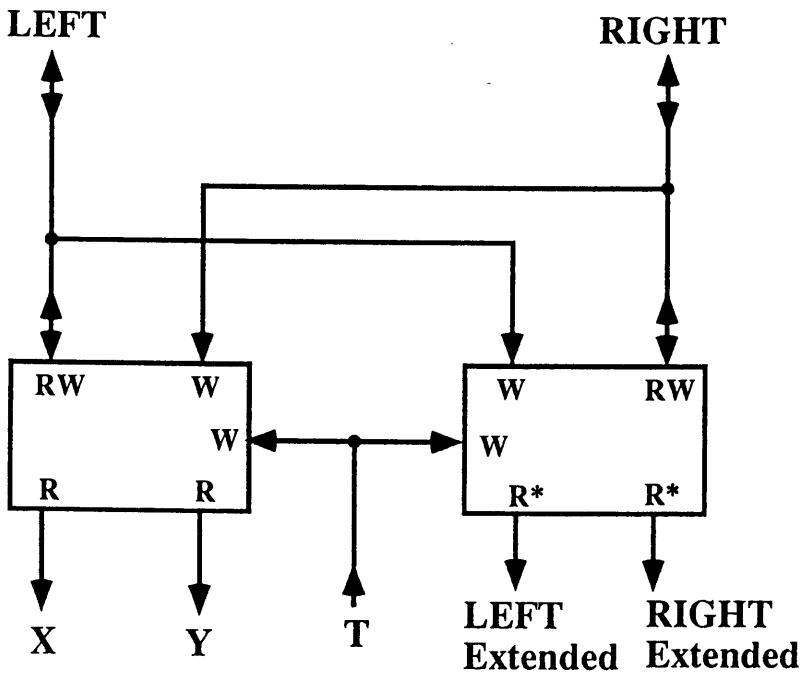
Each dual-B2210A is a pair of chips cascaded to form sixty-four 36-bit wide registers. To obtain 72-bit wide registers, the 36-bit registers are used in pairs, one register holding the most significant half of the word, and the other holding the least significant half. Logically, a dual-B2210A provides thirty-two 72-bit registers, with two read ports, two write ports, and one read/write port, where each port is 36-bits wide.

The register file design, which calls for two dual-B2210s with duplicate contents, arises from the desire to prevent slow memory-write timing from limiting the clock speed of the processor. The math chips respond quickly enough to allow the X, Y, and T buses to each be made only 36-bits wide, and used twice during each cycle in order to transfer a 72-bit quantity. Similarly, the memory system is able to use the LEFT and RIGHT buses twice per cycle to transfer data from the memories to the register file. However, the memory chips are too slow to allow data to written to memory twice per cycle. This requires effectively making the LEFT and RIGHT buses 72 bits wide for register-to-memory transfers. To allow for 72-bit register-to-memory transfers, the two dual-B2210s are connected as shown in figure 4. Whenever data is written into one of the dual-B2210As, it is also written into the other, thereby ensuring that the contents of the two register files is identical. The second register file provides the additional read ports required to access both the the least-significant 36-bits and the most-significant 36-bits of a word simultaneously.

The main buses, the data memories, the register file, the I/O ports, and



(a)



(b)

Figure 4: (a) The central register file is constructed from B.I.T. register-file chips (B2210A) with two read ports, two write ports, and one read/write port. (b) Using two pairs of chips provides the additional read ports (marked with "*" in the figure) required perform 72-bit register-to-memory transfers. These ports carry the lower half of a double precision value while the Left or Right bus carries the upper half.

the data cables are protected by one parity bit for each byte of data. Parity is generated and checked by the math chips.

In addition to the data paths shown in figure 2, there are other buses in the processor. The math chips have a 64-bit internal recirculation path which leads from the output register back to one of the input registers. The output of one math chip can be transferred on the T bus to the input register of the other using this path. Some calculations can be directly chained without going back through the register file. This path can be used, for example, to pass results from the Multiplier directly to the ALU, freeing the X and Y buses to bring in new operands.

There are two 16-bit "side paths" that permit transfers between the address generators and the floating-point buses. One side path links the most significant sixteen bits of the X bus with the left address generator. Similarly, the other side path links the Y bus with the right address generator. The side paths allow floating-point values to be passed to the address generators for use in address computations. In the other direction, the side-path buses permit constants in the address-generator's immediate field to be driven onto the X and Y buses, to be used as operands by the math chips.

1.4 Data Memory

To keep the system balanced with respect to bus bandwidth and to keep the math chips as busy as possible, we chose to have two independent data memories. Each holds 16K double-precision words for a total of 256K bytes on each processor. The data memory is implemented with 20ns 16K $\times$ 4-bit static RAMs. The RAMs are accessed once in each instruction cycle.

Each memory has an associated address generator. Every cycle the address generator produces a new data address which is used in the next cycle to access memory. The address generators are implemented with a simple 16-bit single-chip microprocessor ALU slice (IDT49C402).[2] The chip contains a general purpose ALU backed by a 64-entry address-register file. The address registers can be used to store base addresses or to index through arrays. Immediate values are supplied to the address generators from the instruction word.

[2]This chip and the sequencer chip are 16-bit CMOS versions of the familiar AMD 2901 microprocessor slice and the AMD 2910 microprogram controller.

1.5 Sequencer and Instruction Memory

Overall program flow is controlled by a sequencer chip (IDT49C410). The sequencer produces a new instruction address every cycle, as specified by the current instruction and conditions gathered from elsewhere in the processor. Conditional jumps are based on a number of floating-point flags, the global flag, and a flag from the host processor. The processor is pipelined such that in every cycle, the sequencer calculates the address of the instruction that will be executed two cycles later. Thus a branch instruction issued in cycle N will take effect in cycle $N + 2$. The instruction directly following a jump instruction is always executed.

Each instruction controls the operation of the four main functional units: floating-point, left and right address generators, and sequencer. Instructions are 168 bits (21 bytes) in length, implemented with the same 20ns 16K $\times$ 4-bit parts as the data RAM. A processor can hold 16K instructions or 336Kbytes.

1.6 Input/Output Ports

The I/O ports are connected to the Left and Right buses, along with the the data memories and register file. Each port consists of a double precision register and a very high current (160mA) TTL transceiver (74F30640). The transceiver drives the interboard ribbon cables with the upper and lower half of a data word in a time multiplexed fashion.

Each port transmits a word between processors in two cycles. Interboard communication begins (in cycle 1) when some processor transfers the contents of a register to its I/O port. During cycles 2 and 3, the transmitting port drives the cable with the upper and lower halves of the data word. Also during cycle 3, the receiving processors move the word from the I/O port to a register. Since there is no hardware arbitration on the I/O ports, the programmer must develop a convention for controlling access to each communication channel.

The interboard connections are designed to avoid reflections and achieve a "first wavefront" communication path. To avoid reflections, the transmission lines are wired point to point with no branches. On the board they are routed in a continuous path from an input connector, to the transceiver, and then to an output connector. Cables, connectors, board traces, and terminators all have an impedance of 80 ohms. The ports are implemented with open-

collector drivers to prevent program errors from damaging the hardware.

1.7 Clocks

Clocks for the entire system are generated from a single crystal on a separate clock/host-interface board. Copies of the master clock are carefully buffered and distributed via coax cables to each processor. Clock skew is held to ± 1 ns by using buffers from a single IC package and matched-length cables. This margin is quite adequate for our design.

The processors use a tapped delay line to create controlled clock edges at 7ns intervals. Two programmable logic arrays combine various taps to create a multitude of clock waveforms. Clocks are driven by high-current TTL gates (74F1804) and are parallel-terminated with about 80 ohms. This strategy gave us acceptable control of skews among the various clocks, but the very-high edge rates probably contributed to problems we encountered with clock bounce during debugging.

This clocking methodology proved to be very flexible during the design phase. It promoted reliable design of setup and hold times. A wide variety of clocking requirements for different off-the-shelf parts were easy to accommodate. On the other hand, the large number of clocks were complex to design, modify, terminate, and control.

1.8 Host Interface

Processors communicate with the host workstation via a 16-bit general purpose parallel interface. This is a serial protocol that runs at the speed of the host processor, about 1 μsec for each transaction. Eight of the sixteen bits are used to address one or more processor boards. The remaining eight bits contain data or control information. On the processor, the host interface connects to a 29-byte-long scan path that threads through the instruction register and address registers. It is constructed from 8-bit AMD29818 scannable registers that are daisy-chained together. Shifting data along the scan path provides a bidirectional interface with the host.

This interface is very slow compared to the rest of the machine, and it restricts the useful range of applications of our prototype machine to those in which only a small amount of data is transferred between the processors and the host. This admits significant computations (simulations using ordinary

differential equations), but we are aware of the limitations imposed here. The host interface is one of the first areas of the design that should be improved.

2 Toolkit Low-level Programming Model

The Supercomputer Toolkit processor is programmed as a very-long instruction word (VLIW) computer. The programmer has full control of all of the hardware resources just discussed. Thus, in every cycle, the following operations can be performed in parallel:

- Two floating-point operations, one in the ALU and one in the Multiplier. The ALU and Multiplier share input and output ports so two values can be fetched from the register file and one result can be written back.
- Two memory-I/O bus transactions, one on the Left bus and one on the Right bus. Data is exchanged between memory and the main register file, or between the register file and the I/O port.
- Two address computations, one in the left address generator and one in the right address generator. The addresses will be used to access the data memories in the following cycle. The address generators have internal register files to support these operations.
- One sequencer operation, to generate an instruction address. Typical sequencer instructions include conditional branch, jump, continue, and call.
- Two flags may be set: the global flag goes to neighboring processors; the host flag goes to the host workstation.

To program the Toolkit at this level, we use a primitive symbolic assembly language. An instruction appears as a list of up to six tagged fields:

```
((flop ...)
 (lmio ...) (rmio ...)
 (lag ...)  (rag ...)
 (sequencer ...)
 (flags ...))
```

The fields may be listed any order. The assembler supplies default values for omitted fields.

2.1 Floating-point Operations

The ALU and Multiplier operate simultaneously. Each Toolkit instruction has a separate opcode field for the ALU and Multiplier. Any ALU opcode can appear together with any Multiplier opcode. Most floating-point, integer, and logical operations require one cycle. Divide requires 4 cycles and square root requires 7 cycles. The entire list of available operations can be found in the B.I.T. data sheet for this chip set [5].

The following six-cycle sequence illustrates the pipelining of these operations, as well as the assembler syntax for the floating-point portion of a Toolkit instruction. As illustrated here and in the examples to follow, the processor pipeline is controlled partly in hardware and partly in software. For example, floating-point opcodes are specified in the same instruction as the operand register numbers. Pipeline registers hold and delay the opcodes while the register read takes place. Then, the opcodes and operands enter the math chips together on the following cycle. The command to latch the math chip result register (`&latch`) is specified in another instruction, and the command to write the result back to the register file is specified in a third (`t ...`).

```
Cycle                    Instruction

  1      ((flop (x r10) (y r11) (* dmult x y)))
  2      ((flop (x r12) (y r13) (* dmult x y &latch)
                                (+ dadd x y)))
  3      ((flop (x r14) (y r15) (* dmult x y &latch)
                        (t * r5) (+ dadd t z &latch)))
  4      ((flop (* &latch) (+ &latch)))
  5      ((flop (t + r6)))
  6      ((flop (t * r7)))
```

The instruction in cycle 1 places the contents of r_{10} on the X bus, the contents of r_{11} on the Y bus, and passes the double-precision multiply opcode to the Multiplier.

In cycle 2, the multiplier computes the product requested in cycle 1, and then latches the result into its result register. Also during cycle 2, instructions and data are passed to the Multiplier and ALU requesting both the product $r_{12}r_{13}$ and the sum $r_{12} + r_{13}$.

In cycle 3, the product $r_{10}r_{11}$ is driven from the Multiplier onto the T bus and is written into r_5. This value is also specified as an operand for the next ALU operation. Simultaneously, the product $r_{12}r_{13}$ and the sum $r_{12} + r_{13}$ are computed and latched into the result registers of the Multiplier and ALU. Data for the next multiplication is transferred from registers r_{14} and r_{15}. The second ALU operand, $r_{12} + r_{13}$, is transferred over the ALU's internal feedback path from the result register back to an operand register.

In cycle 4 the new sum $(r_{10}r_{11} + r_{12} + r_{13})$ is computed and latched by the ALU. The Multiplier computes and latches $r_{14}r_{15}$.

In cycle 5, the ALU result is written to r_6 via the T bus.

In cycle 6 the Multiplier result computed and latched during cycle 4 is written to r_7.

2.2 Bus Operations

Each of the two memory-I/O buses can perform a 64-bit transfer either between registers and memory, or between registers and an I/O port. There are no direct register to register operations or memory to memory operations. Register-memory transfers require an address to have been generated during the previous cycle. For example, the instruction

```
((lmio m->r r23) (rmio r->m r17))
```

loads r_{23} with the contents of the left-memory address and stores r_{17} into the right memory address.

2.3 Input/Output Operations

Communication between boards is accomplished by transferring 64-bit quantities between registers and the I/O ports. For example, suppose that the left I/O port of Processor 1 is connected to the right port of Processor 2 and to the left port of Processor 3. This code fragment transmits the contents of r_{10} in Processor 1 to r_{20} in Processor 2 and to r_{25} in Processor 3:

```
Cycle   Processor 1       Processor 2      Processor 3

  1   (lmio r->io r10)         ...              ...
  2          ...               ...              ...
  3          ...         (rmio io->r r20)  (lmio io->r r25)
```

2.4 Address Generation

Each of the two address generators produces a new result every cycle, which is used as the memory address for the following cycle. Each address generator has its own internal ALU, and 64 16-bit registers to store addresses. Each operation may take its sources from the registers, from a 16-bit immediate field in the instruction word, or from the main register file via the side paths.

ALU operands are denoted by A and B, the constant field by D, a zero source by Z. Available ALU operations include addition, subtraction, and Boolean operations. The result of an operation can be passed to the address generator output, and may be written back to the internal address-register selected by B.

The syntax of the address-generator field of a Toolkit instruction is shown in figure 5. Here are a few examples:

`(lag (add dz nop low) (d 1117))` The left address generator performs the addition of D and Zero with carry-in set low. The result, 1117, is passed to the output, without writing it back to the address-register file (`nop`).

`(rag (add zb ramf high) (b ag-r0))` This performs pre-increment indexed addressing using register `ag-r0` as the index register. The right address generator increments the contents of `ag-r0` by adding it to zero with the carry-in set high. The incremented result becomes the address-generator output and is stored back into `ag-r0` as specified by the destination operation `ramf`.

`(rag (add zb rama high) (a ag-r0) (b ag-r0))` This performs post-increment addressing. It increments the contents of the B address-register and stores the result in the B address-register, thereby updating the contents of `ag-r0`. This time, the `rama` operation specifies that the address

```
<field> ::= ( <lag | rag> <ag-op> )

<ag-op> ::= ( <source-op> <sources> <dest-op> <carry> )
              <a-value> <b-value> <d-value>

<source-op> ::= add | or | and | exor | ...

<sources> ::=  ab | zb | za | da | dz | ...

<dest-op> ::= nop | ramf | rama | oreg | ...

<carry> ::= low | high

a-value ::= ( a <ag-register 0 - 15> )

b-value ::= ( b <ag-register 0 - 15> )

d-value ::= ( d <constant 0 - 65535> )
```

Figure 5: Syntax of the address-generator fields. A full list of operations can be found in the documentation for the address generator chip, IDT49C402 [7].

generator output should be the A-source, which is the (unincremented) contents of `ag-r0`.

2.5 Instruction Sequencing

The sequencing operations [7] include conditional branches, subroutine calls, and looping, all maintained through a 33-deep on-chip stack. For example, loops are implemented using the sequencer operations `push` and `rfct`. `Push` pushes the next address onto the internal stack, and sets an internal counter to the number of loop iterations minus one. `Rfct` decrements the counter and keeps branching to the saved address until the count becomes negative, at which point it pops the stack and falls through the loop. Sequencer operations are pipelined and take effect one cycle after they appear in the instruction stream. For example, the first instruction in the body of a `push/rfct` loop will be two cycles after the appearance of the `push` in the instruction stream.[3]

Figure 6 shows a subroutine that uses the loop instructions to upload to the host a known-length vector by repeatedly calling the `upload` subroutine, which uploads the floating-point number in `r1`. The vector is stored in consecutive locations in left memory. The caller initializes the left address-register named `ptr` to point to the first of these locations. The vector length minus one is the constant `n-1`, which must be known at assembly time.

The program example shown in figure 7 embeds `upload-vector` in an outer loop to produce a routine that uploads all elements of an $n \times n$ matrix (where n is known at assembly time). The inner loop (over the elements of a row) is counted, as above, in the sequencer. The outer loop (over the rows) is counted by incrementing (with the ALU) the register `count-up-rows`, which is initialized to $-n$, and branching back so long as the result is negative. In keeping with the convention that the ALU is used for floating-point operations, we assume that the initial (floating-point) value $-n$ is stored in a known location `address-of-minus-n`, and that `one` is a register containing the (floating-point) value 1. We assume that the matrix is stored so that M_{ij} is at a base address plus $i \times$row-increment$+j$ where row-increment is a constant specified at assembly time.

[3]The instruction directly following a jump is referred to as the "jump slot" in modern reduced instruction set computers.

```
Cycle
     (label upload-vector)
1    ((sequencer push true n-1))          ;do loop n times
     ;;Generate address for first vector element during the
     ;;delay slot for the push:
2    ((lag (add zb nop low) (b ptr)))

     ;;Body of loop is cycles 3 through 6
     ;;Move vector element to R1
3    ((lmio m->r r1)
      (sequencer cjs true upload))        ;subroutine call to upload
4    ()                                   ;delay slot for the cjs
5    ((sequencer rfct))
     ;;increment the pointer during the delay slot for the rfct:
6    ((lag (add zb ramf high) (b ptr)))

7    ((sequencer crtn true))              ;return from upload-vector
8    ()                                   ;delay slot for the crtn
```

Figure 6: A Toolkit subroutine to upload a vector to the host, using a simple loop counter in the sequencer. Note in this example that the sequencer operations **push**, **rfct**, **cjs** (conditional jump to subroutine), and **crtn** (conditional return from subroutine) are here all conditionalized with **true**, so that they always take effect.

The routine is called with the left-address-generator register `matrix-base` pointing to M_{00}. It uses left-address-generator registers `row-pointer` to hold a pointer to the beginning of successive rows, `ptr` to hold the pointer to the individual elements M_{ij}.

The `upload-matrix` program uses a conditional jump based upon a flag that signals whether the floating-point result is negative. Flags are set by the ALU or the Multiplier when they drive the T bus.

2.6 Multiprocessing and Synchronization

Since all Toolkit processors are driven by a single master clock, the individual processors are electrically synchronized in terms of when cycle boundaries occur. However, each processor has its own sequencer and instruction memory, so the operations performed by the processors are, in general, independent of one another. When data is transferred among processors the programmer must arrange that the receivers all grab the data two cycles after the transmitter puts it out. This can be passively arranged by cycle counting or by an explicit synchronization action.

The one-bit `global` flag can be used to maintain synchronization among processors during the course of a computation. One programming technique is to maintain synchronization between processors at all times by having each processor execute the same instruction sequence: All processors branch simultaneously, based on the condition being asserted on the `global` flag, in effect, behaving as if there was one central controller governing all processors. This approach works well on vectorizable, data-independent problems, but breaks down when local data-dependent decisions must be made. A more complex alternative is to allow each processor to execute branches independently based on its own local data, with synchronization occurring only occasionally when communication is required.

For example, a step in a multiprocessor program might require each processor P_i to perform a computation C_i, where different C_i may require different numbers of cycles. In this case, the `global` flag could be used as a "busy" indicator: As long as each processor is busy, it asserts the flag. When all processors are done, they release the flag, informing all processors that they can proceed to the next phase of the computation. The subroutine `wait-for-all-boards`, shown in figure 8, can be used to implement this kind of synchronization. Any processor calling this routine will remain in a wait

```
(label upload-matrix)
;; Initialize outer-loop by initializing row-pointer in the lag
;; to the start of the first row.  In the same cycle, use the rag
;; to generate the address for initializing the
;; count-up-rows register.
((rag (add dz nop low) (d address-of-minus-n))
 (lag (add za ramf low) (a matrix-base) (b row-pointer)))
;; Initialize count-up-rows
((rmio m->r count-up-rows))

(label outer-loop)
((sequencer push true n-1))          ;ready to count the inner loop
;; Initialize ptr to row-pointer
((lag (add za ramf low) (a row-pointer) (b ptr)))

;;Next four cycles form the inner loop, as in upload vector
((lmio m->r r1) (sequencer cjs true upload))
()
((sequencer rfct))
((lag (add zb ramf high) (b ptr)))

;;Increment count-up-rows, using the ALU
((flop (x count-up-rows) (y ,one) (+ dadd x y)))
;;Latch the result.  In the same cycle, increment row-pointer in
;;the lag to point to the start of the next row.
((flop (+ &latch))
 (lag (add da ramf low) (a row-pointer) (b row-pointer)
                        (d row-increment)))
;;Store the new value of count-up-rows, and jump back to the
;;start of the outer loop if the result is still negative
((flop (t + count-up-rows))
 (sequencer cjp neg row-loop))
()                                   ;delay slot for the cjp

((sequencer crtn true))              ;return from upload-matrix
()                                   ;delay slot for the crtn
```

Figure 7: This Toolkit program uploads a matrix, using nested loops.

```
Cycle
        (label wait-for-all-boards)
1       ((sequencer ldct true wait-for-all-boards-done))
2       ((sequencer cjp true wait-for-all-boards-assert-ready))
        (label wait-for-all-boards-assert-ready)
3       ((sequencer jrp global wait-for-all-boards-assert-ready)
         (flags (global . low)))
        (label wait-for-all-boards-done)
4       ()
5       ((sequencer crtn))
6       ()
```

Figure 8: This routine uses the `global` flag to synchronize processors so that they will all return in the same cycle when the flag is deasserted by all boards. The Toolkit assembler default for the `flags` field asserts the `global` flag, ensuring that it will remain asserted until all processors execute the instruction in cycle 3. As each processor calls the routine, it waits at (3) in a 1-cycle loop and deasserts the flag. The loop is implemented with the sequencer `jrp` conditional jump instruction, which branches either to the address specified in the instruction, or to the address stored in the internal counter, depending on the state of the flag. The counter register is loaded by the `ldct` instruction in cycle 1. Note that a one-cycle loop is attained even though the sequencer is pipelined, by including a jump instruction (3) in the "jump slot" of a previous jump (2). Interested readers should trace through the control structure here in detail, noting that when the loop terminates, the no-op at 4 is executed twice.

loop until all processors have called the routine, whereupon all processors return in the same cycle.

2.7 Benchmark Examples

Figure 9 shows a single-processor program that computes the inner product of two vectors at the processor's peak speed of two floating-point operations per cycle (23.8 double-precision MFLOPS at the current clock rate). The two vectors are stored in memory, one in the left memory and one in the right memory. The routine is pipelined to use a two-cycle loop that does two multiplications and two additions.

```
((sequencer push true n-1)                ; vector length = 2n
 (flop (x r10) (y r10) (* dmult x y))     ; c(r10)=0
 (lag (add dz ramf high) (d lbase-1) (b a1))
 (rag (add dz ramf high) (d rbase-1) (b a1)))

((flop (x r10) (y r10) (* dmult x y &latch) (+ dadd x y))
 (lmio m->r r1) (rmio m->r r2)
 (lag (add zb ramf high) (b a1))
 (rag (add zb ramf high) (b a1)))

;; The next two cycles are the inner loop.
((sequencer rfct)
 (flop (x r1) (y r2) (t *)
       (* dmult x y &latch) (+ dadd t z &latch))
 (lmio m->r r3) (rmio m->r r4)
 (lag (add zb ramf high) (b a1))
 (rag (add zb ramf high) (b a1)))

((flop (x r3) (y r4) (t *)
       (* dmult x y &latch) (+ dadd t z &latch))
 (lmio m->r r1) (rmio m->r r2)
 (lag (add zb ramf high) (b a1))
 (rag (add zb ramf high) (b a1)))

((flop (t *) (* &latch) (+ dadd t z &latch)))
((flop (t *) (+ dadd t z &latch)))
((flop (+ &latch)) (sequencer cjp true done))
((flop (t + r5))))
(label done)
```

Figure 9: This routine computes the inner product of two vectors of length $2n$ at the processor's peak speed of 2 floating-point operations per cycle. One vector is stored in left memory beginning at location 1+lbase-1, the other in right memory beginning at location 1+rbase-1. The inner loop, which does two multiplications and two additions, is executed n times.

Another Toolkit demonstration program (code not shown here) solves $n \times n$ systems of linear equations $Ax = b$ by means of Gauss-Jordan elimination with full pivoting, using the algorithm given in [12]. The running time of the algorithm is dominated by the row reduction performed each time a pivot is selected. This involves subtracting a multiple of the pivot row from each other row of the matrix:

```
DO 21 LL=1,N
  IF (LL.NE.ICOL) THEN
      DUM=A(LL,ICOL)
      A(LL,ICOL)=0.0
      DO 18 L=1,N
        A(LL,L)=A(LL,L)-A(ICOL,L)*DUM
        18 CONTINUE
      B(LL)=B(LL)-B(ICOL)*DUM
  ENDIF
21 CONTINUE
```

The Toolkit implementation runs the inner loop (DO loop 18) at a rate of one floating-point operation per cycle (11.9 MFLOPS). This is accomplished by holding the matrix A in left memory, but copying the pivot row A[ICOL,*] into right memory when the pivot is chosen. Using both memories provides enough bandwidth to schedule a floating-point operation on every cycle of the inner loop.

The bottleneck in this computation is the shared floating-point result bus which is shared by the Multiplier and ALU. Thus, even though subtractions and multiplications could be done simultaneously, only a single result can be written to the registers on each cycle. In contrast, the inner-product program can sustain two floating-point operations per cycle because it uses the ALU result register to hold the partially computed sum, and writing to the registers is not required.

3 High-level Programming Model

The Toolkit software environment includes a compiler that converts numerical routines written in a high-level language (the Scheme dialect of Lisp [6]) into extremely efficient, highly pipelined Toolkit programs. These compiled

routines can be combined with hand-written assembly language programs by means of a programmable linker.

The Toolkit compilation methodology requires the programmer to divide programs into data-independent computations. A data-independent computation is one in which the sequence of operations is not dependent on the particular numerical values of the data being manipulated. For instance, the sequence of multiplications and additions performed in a Fast Fourier Transform is independent of the numerical values of the data being transformed. The compiler generates Toolkit instructions for a program's data-independent computations, leaving the programmer to implement the data-dependent branches in assembly language.

3.1 Compiling Data-independent Computations

The first stage of the compilation process uses the partial evaluation technique described in [3] and [4] to extract the underlying numerical computation from the high-level program. Unlike high-level language programs which may include procedure calls and complex data structure manipulations, the optimized programs produced by partial evaluation consist entirely of numerical operations. This purely numerical program is further optimized using traditional compiler techniques such as constant-folding, dead-code elimination, and common-expression elimination to produce a graph of numerical operations to be performed.

The second stage of the compiler maps the graph onto the particular architecture of the Toolkit processor to generate actual Toolkit instructions. This process has two phases. First, the compiler chooses a preliminary order of operations to reduce memory traffic with no regard for actual pipeline delays. This order serves as advice to the second phase, attempts to fill all available slots in the pipeline. It also imposes some other constraints. For example, if two operands are to be loaded at the same time, the compiler attempts to assign them to different memory banks.

This two-phase strategy seems to give very good performance. It combines traditional "results-up scheduling," in which the data-flow graph of the computation is analyzed to determine the ordering of the instructions that will maximize the immediate reuse of intermediate results, with "cycle-based scheduling," which works forward through the program from the operands towards the results, choosing the order of the instructions incrementally in

an attempt to keep the processor pipeline full.

Compiled programs bear little resemblance to hand-coded programs. In hand-written programs, pipeline stages relating to a particular operation, such as a multiply, tend to be located close to each other. In contrast, the compiled code spreads computations out over time in order to fill in pipeline slots. The compiler schedules loads and stores retroactively, intentionally placing them as far back in time as possible, to leave memory access opportunities available for later instructions. Indeed, loading a register from memory may occur dozens of cycles before the operand is used. Partial evaluation makes this extreme lookahead possible by providing huge basic blocks of straight-line numerical code. The basic blocks produced by the partial evaluator are so large that any non-trivial register allocation technique could be expected to do well.

The compiled code tends to be extremely efficient, even when generated from very high-level source code. As a simple example, figure 10 shows a highly abstract definition of a vector addition procedure `add-vectors`, implemented by applying to the addition operator a general transformation `vector-elementwise`, which converts an n-ary scalar function f into an n-ary vector function that applies f to the corresponding elements of n-vectors, $v^1 = (v_1^1, v_2^1, \ldots), v^2 = (v_1^2, v_2^2, \ldots), \ldots, v^n = (v_1^n, v_2^n, \ldots)$, and produces the vector of results $(f(v_1^1, \ldots, v_1^n), f(v_2^1, \ldots, v_2^n), \ldots)$. This implementation of `add-vectors` is inefficient in most Scheme implementations. Since the procedures operate on vectors of any length, there must be a run-time loop that counts through the vector index (here implemented in `generate-vector`). An even greater source of inefficiency is the fact that the arity of the procedure argument `f` is not known to `vector-elementwise`, which therefore must explicitly construct lists for `f` at run time. The compilation is specified by applying `add-vectors` to two vectors, each consisting of four placeholders (see [3]). The placeholders represent the inputs to the compiled program. Specifying the number of vectors and vector length permits the compiler to generate code after all tests and operations on data structures are performed at compile time. This leaves only the actual component additions as the only "real work" to be performed at run time.

Figure 11 shows the compiled output. The entire computation, including moving data to and from memory, is accomplished in nine cycles. This density of "useful" operations—4 additions in 9 cycles—is untypically low for compiler output, because the program is so short.

```
(define (vector-elementwise f)
  (lambda (vectors)
    (generate-vector
     (vector-length (car vectors))
     (lambda (i)
       (apply f (map (lambda (v) (vector-ref  v i))
                     vectors))))))

(define (generate-vector size proc)
  (let ((ans (make-vector size)))
    (let loop ((i 0))
      (if (= i size)
          ans
          (begin (vector-set! ans i (proc i))
                 (loop (+ i 1)))))))

(define add-vectors (elementwise +))

(define vector-1
  (vector (make-placeholder 'vector-1-element-1)
          (make-placeholder 'vector-1-element-2)
          (make-placeholder 'vector-1-element-3)
          (make-placeholder 'vector-1-element-4)))

(define vector-2
  (vector (make-placeholder 'vector-2-element-1)
          (make-placeholder 'vector-2-element-2)
          (make-placeholder 'vector-2-element-3)
          (make-placeholder 'vector-2-element-4)))

(add-vectors vector-1 vector-2)
```

Figure 10: This highly abstract implementation of `add-vectors` works for an arbitrary number of vectors of arbitrary length. Using placeholders, the definition is specialized to add two vectors of length four.

```
((flop) (lag (add dz nop low) (d 5)) (rag (add dz nop low) (d 1)))
((flop)
 (lag (add dz nop low) (d 7))
 (rag (add dz nop low) (d 3))
 (lmio m->r r20)
 (rmio m->r r19))
((flop (+ dadd x y) (x r20) (y r19))
 (lag (add dz nop low) (d 4))
 (rag (add dz nop low) (d 0))
 (lmio m->r r3)
 (rmio m->r r26))
((flop (+ &latch dadd x y) (x r3) (y r26))
 (lag (add dz nop low) (d 6))
 (rag (add dz nop low) (d 2))
 (lmio m->r r9)
 (rmio m->r r8))
((flop (+ &latch dadd x y) (x r9) (y r8) (t + r20))
 (lag (add dz nop low) (d 8))
 (rag (add dz nop low) (d 8))
 (lmio m->r r19)
 (rmio m->r r17))
((flop (+ &latch dadd x y) (x r19) (y r17) (t + r3))
 (lag (add dz nop low) (d 9))
 (rag (add dz nop low) (d 9))
 (lmio r->m r20)
 (rmio r->m r20))
((flop (+ &latch) (t + r9))
 (lag (add dz nop low) (d 10))
 (rag (add dz nop low) (d 10))
 (lmio r->m r3)
 (rmio r->m r3))
((flop (t + r19))
 (lag (add dz nop low) (d 11))
 (rag (add dz nop low) (d 11))
 (lmio r->m r9)
 (rmio r->m r9))
((flop) (lmio r->m r19) (rmio r->m r19))
```

Figure 11: This is the compiled output for the program shown in figure 10.

Figure 12 shows a straightforward recursive implementation of the FFT, based upon some suitable representation of complex arithmetic (for example, representing a complex number as a list of its real and imaginary parts). Even though the program is simple, written with almost no concessions to efficiency, the compiler transforms this into highly efficient code. A 128-point FFT requires 4222 cycles (0.354 msec. at our clock rate), performing 3716 floating-point operations, plus the required moves to and from memory.[4] This is still a rather low density of floating-point operations, compared with our experience with ordinary differential equations (see section 4). For the FFT computation in particular, one can do much better than this by using the ALU and Multiplier together in inner product computations. The present compiler, however, does not schedule the math chips simultaneously. Given the contention for the floating-point result bus, we do not expect to obtain a significant speed improvement in general. Inner product computations, of course, are an important exception. See the comments in section 2.7.

3.2 The Programmable Linker

Currently, the compiler handles only straight-line code, and permits calls to a library of hand-coded numerical subroutines. This restriction is severe but tolerable for many important applications—many numerical programs consist of long segments of straight-line code separated by just a few run-time tests. For example, integrating a system of ordinary differential equations over an interval with an explicit-formula integrator such as Runge-Kutta requires run-time tests only to determine if the end of the interval has been reached, and, for an adaptive integrator, whether to adjust the step-size. In contrast, an implicit-formula integrator such as Backward Euler requires the solution of systems of linear equations to choose good pivots, thereby generating data-dependent operations. This is currently beyond the capability of the compiler, and the Toolkit numerical subroutine library includes a hand-coded linear-equation solver that can be used in conjunction with compiled code.

Combining compiled and hand-written assembly code is accomplished with a programmable linker. As simple example, we show how to construct Toolkit program that integrates a system of ordinary differential equations

[4] A 128-point FFT is small, but the compiled block can now be called as a subroutine in computing larger FFTs.

```
(define (fft data-list)
  (let ((roots (roots-of-unity (length data-list))))
    (define (internal-fft data-list)
      (let ((n (length data-list)))
        (cond ((= n 1) data-list)
              ((odd? n) (error "FFT input not a power of 2"))
              (else
               (let ((even-terms (internal-fft (evens data-list)))
                     (odd-terms (w* (internal-fft (odds data-list))
                                    roots)))
                 (append (map complex+ even-terms odd-terms)
                         (map complex- even-terms odd-terms)))))))
    (internal-fft data-list)))

(define (w* data roots)
  (if (null? data)
      '()
      (cons (complex* (car data) (car roots))
            (w* (cdr data) (cdr roots)))))

(define (evens list)
  (if (null? list)
      '()
      (cons (car list) (evens (cddr list)))))

(define (odds list)
  (if (null? list)
      '()
      (cons (cadr list) (odds (cddr list)))))
```

Figure 12: A straightforward FFT implementation, which is transformed by the compiler into efficient Toolkit code.

using Runge-Kutta integration. The program, running on the Toolkit, reads from the host an initial state and the number of steps to integrate, performs the integration, and uploads the final state to the host. This is all straight-line code except for counting the number of steps.

Figure 13 shows the Scheme source code to be compiled. (The details of the procedure `ode-system` are not shown—this is any procedure that accepts a state vector and generates the vector of derivatives.) Even though the formulation of the Runge-Kutta integrator is in terms of abstract vector operations, this will exact no performance penalty. The compiler uses the specification of the input placeholder data structure to specialize all operations, in this case, to vectors of length four. The compiler output, which integrates the system for one step, is generated by the call to `compile-kit-program` (shown in the figure) which combines the input placeholder data structure with the Scheme program that defines the computation.

Figure 14 defines some address and data structures used by the linker to handle the compiled output. The first four `define` expressions cause the linker to allocate memory for the number of integration steps, for a counter, and for the constants 0 and 1. The next two `define` expressions allocate storage for the inputs and outputs of the compiled computation. Figure 15 shows how the compiler output is included as a subroutine with hand-written assembly code. The `define-program` expression generates the program text that will be passed to the assembler. After some initialization, the program downloads the number of integration steps and the initial state to the Toolkit, and runs a simple loop that calls the compiler-generated `integrate-one-step` subroutine. The assembly code is written in a mixture of primitive assembly instructions and simple assembly-language macros.

Note that this assembled Toolkit program could be loaded onto multiple Toolkit boards, producing a system that will perform multiple integrations simultaneously, starting from different initial states.

It is amusing to observe that this style of pasting together hand-written and compiled code inverts the traditional role of high-level and low-level programming. Ordinarily, one writes code in a high-level language that may call hand-written assembly-code subroutines for speed-critical applications. Here, we write the inner-loops in a high-level language, and compile them for inclusion as subroutines, called from hand-written assembly code.

```
;;;Runge-Kutta-4 integrator
(define (rkstep f state h)
  (let* ((h* (scale-vector h))
         (k0 (h* (f state)))
         (k1 (h* (f (add-vectors state (1/2* k0)))))
         (k2 (h* (f (add-vectors state (1/2* k1)))))
         (k3 (h* (f (add-vectors state k2)))))
    (add-vectors state
                 (1/6* (add-vectors k0 (2* k1) (2* k2) k3)))))

;;; This is any function that takes a state vector
;;; and returns the vector of derivatives.
(define (ode-system state) ...)

;;; We specify the stepsize here.
(define (integrate-one-step state)
  (rkstep ode-system state 1.e-6))

;;; This placeholder structure specializes the computation
;;; to a four-dimensional state.
(define input-placeholders
  (vector (make-unknown 'x0)
          (make-unknown 'x1)
          (make-unknown 'x2)
          (make-unknown 'x3)))

(compile-kit-program
 "integrate-one-step"  ;name used by linker for the resulting program
 input-placeholders    ;input structure
 integrate-one-step    ;Scheme definition of program to compile
 true)                 ;this flag defines automatically generates code
                       ;that copies the computation's outputs back to
                       ;its inputs -- handy for loops
```

Figure 13: This Scheme definitions generate compiled the code for `integrate-one-step`, which performs one step of a fourth-order Runge-Kutta integration. The `ode-system` is any Scheme procedure that accepts a vector as argument and returns the corresponding vector of derivatives. The input placeholder data structure causes the compiler to specialize the program to vectors of length four.

```
(define integrator-limit-address (allocate-dual-memory 1))
(define integrator-counter-address (allocate-dual-memory 1))

(define zero-source (dual-memory-constant-location 0.0))
(define one-source (dual-memory-constant-location 1.0))

(define integrate-one-step-inputs
  (computation-input-data-structure-with-address-pointers
   integrate-one-step))

(define integrate-one-step-outputs
  (computation-output-data-structure-with-address-pointers
   integrate-one-step))
```

Figure 14: Definitions used by the linker in handling the compiled result of the source code shown in figure 13.

4 Example: A Breakthrough in Solar-System Integrations

Even though our prototype Toolkit has been operational for only a few months, we have already used it to perform a computation of major scientific significance—an integration of the complete Solar System that improves upon previous computations by nearly two orders of magnitude.

One of our motivating examples in embarking on the Toolkit project was our experience with the Digital Orrery [2]. The Orrery, constructed in 1983-1984, is a special-purpose numerical engine optimized for high-precision numerical integrations of the equations of motion of small numbers of gravitationally interacting bodies. Using 1980 technology, the device is about 1 cubic foot of electronics, dissipating 150 watts. On the problem it was designed to solve, it is measured to be 60 times faster than a VAX 11/780 with FPA, or 1/3 the speed of a Cray 1S.[5] In 1988, Sussman and Wisdom [11] used the Orrery to demonstrate that the long-term motion of the planet Pluto, and by implication the dynamics of the Solar System, is chaotic. The positions of the outer planets were integrated for a simulated time of 845 million

[5]With the arrival of the Supercomputer Toolkit, the Orrery was officially retired, and was transferred to the Smithsonian Institution in August, 1991.

```
(define-program
  '(
    ((sequencer cjp true start-of-program))
    ()
    ;;include standard libraries here
    ...
    ;;include the compiler output as a subroutine
    (label integrate-one-step)
    ,@(include-computation integrate-one-step)
    ((sequencer crtn true))
    ()
    (label start-of-program)
    ;;code that initializes various constants defined by the linker

    (label restart)
    ;;download number of steps from the host
    ,@(dynamic-download-both-memories integrator-limit-address)
    ;;download the initial state
    ,@(dynamic-download-data-structure integrate-one-step-inputs)

    (label integrator-restart)
    ,@(dual-memory-copy
       (make-dual-copy-specifier zero-source
                                 integrator-counter-address))
    (label integrator-loop)
    ,@(jump-if-equal integrator-limit-address
                     integrator-counter-address
                     'return-answer)
    ;;otherwise, increment counter by one:
    ,@(dual-memory-falu-memory-operation 'DADD
                                         one-source
                                         integrator-counter-address
                                         integrator-counter-address)
    ,@(cjs 'integrate-one-step)
    ,@(jump 'integrator-loop)

    (label return-answer)
    ,@(dynamic-upload-data-structure integrate-step-outputs)
    ,@(jump 'integrator-restart)))
```

Figure 15: The compiled code generated by the program in figure 13 is combined with hand-generated assembly code, producing a program that integrates a system of ODEs for a given number of steps, starting with an initial state.

years, a computation in which the Orrery ran continuously for more than three months. Before the Orrery, high-precision integrations over simulated millions of years were prohibitively expensive, and astrophysicists had done only a few small experiments using carefully scheduled resources.

It was natural, then, that our first task for the new Toolkit was to duplicate some of the Orrery's outer planet integrations. This allowed us to check out and debug the Toolkit on a real problem. We quickly implemented a high-order multistep integrator of the Stormer type—written in Scheme and compiled using the Toolkit compiler—and discovered that each board of the Toolkit was about three times faster than the entire Orrery on this program. Some of this speedup was because the Orrery did not have a single instruction to compute square root or divide. In the Orrery, such operations were performed by Newton's method, using a table to get initial approximations.

Wisdom and Holman [13] then developed a new kind of symplectic integrator for use in Solar-System integrations. This integrator allows the particles to advance as if they were only in the field of the Sun. It then corrects their velocities by adding the change in momentum accrued over the drift time by perturbations of the other planets. This integrator is about a factor of ten faster than the traditional Stormer's method, but it is not as accurate over short timescales. Over long timescales it appears not to accrue too much error.

With the speedups available from the Toolkit and the new integrator, we performed a 100-million-year integration of the entire Solar System (not just the outer planets, as with the Orrery), incorporating a post-Newtonian approximation to General Relativity and corrections for the quadrupole moment of the Earth-Moon system. The longest previous such integration [10], recently published, was for about 3 million years.

The results of our integration verify the principal results of the Orrery integrations. We also found good evidence for chaotic motions in the inner Solar System. In particular, we found a Lyapunov exponent for the inner Solar System similar to that predicted by Laskar [8] in his integrations of the averaged equations of motion. We also verified Laskar's prediction that certain pendulum-like resonant variables alternate between circulation and libration. A more complete analysis of the data from the integration will be forthcoming.

Our integration used eight Toolkit boards running in parallel. Each board ran a Solar System with slightly different initial conditions. The evolving

differences were compared to estimate the largest Lyapunov exponent. The differences in initial conditions were chosen to be 1 part in 10^{16} in one coordinate of a planet. We found that, the chaotic amplification of this difference is such that a 1 cm difference in the position of Pluto at the beginning of the integration produces a 1 Astronomical Unit difference in the position of the Earth at the end of the integration.

Programming the integration proceeded essentially along the lines of the simple Runge-Kutta example presented in section 3.2. The integrator and the force law were written as high-level Scheme programs. The accumulation of position was implemented in quad precision (128 bits), and the required quad precision operators were written in Scheme.[6] The Scheme source was compiled with the Toolkit compiler, and the resulting routines were combined with a small amount of Toolkit assembly code, similar to figure 15. [7] The compiled code contains almost 10,000 Toolkit cycles for each integration step, and more than 98 percent of the cycles perform floating-point operations.

The host-side control program was also written in Scheme. The Toolkit was downloaded with initial conditions. It was then repeatedly run for 10^6 7.2-day integration steps, with the state uploaded to the host at the end of each 10^6 step segment—this took about 12 minutes of real time. The 100 million year integration took about 5000 such segments, for a total time of about 1000 hours of run time.

5 Conclusions

In hindsight, there are a few places where the architecture could be improved. Most serious is the fact that there are no datapaths that allow results to be stored in code memory, so that, for example, all repeat counts in the sequencer must be fixed at compile time. Even worse, the sequencer chip provides no method of to get at the addresses stored in its internal registers

[6] In hindsight, the use of quad precision appears to have been overly conservative for this problem, and we plan to rerun the computation at ordinary double precision to confirm this.

[7] Instead of running a separate Solar System on each board, we could have used an alternative parallel decomposition of the problem into one body per board, as was done with the Digital Orrery. In that case, the hand-generated assembly code would have been slightly different, because we would have to copy state information among boards, but the bulk of the code generation would have still been done by the compiler.

without executing the instructions at those addresses. This makes it very difficult to catch and process interrupts, e.g., those caused by parity errors. A better design would make that state more accessible. It would be convenient to set and sense condition codes based upon results of address-generator computations. Also, the ALU and Multiplier share a single result bus which limits the opportunities to use these units simultaneously. But these are mostly annoyances rather than significant problems.

A more serious limitation of our prototype Toolkit is that we have provided only slow communication with the host. This limits applications to those that require very little communication, such as the long-term integration of systems of ordinary differential equations. It is relatively easy to improve this by fabricating a special board with connections to the fast interprocessor communication channels. Such a communications adapter could buffer communications to and from the host, at host-memory speed. Improving this communication is a top priority for future hardware development.

Our software-support system also has a long way to go. While the compiler makes it easy to compile straight-line segments of code, automating the translation of large algebraically-specified systems such as force laws, it has no concept of data-structure or of data-dependent conditional jump. Thus, for example, all of the complex control structures that support variable step size and variable order integrators (such as Gear's method for stiff systems), and even the process of selecting a pivot in a linear equation solver, must currently be painfully constructed in assembly language. The resulting assembly code must be combined with compiler output for the straight-line portions of the computation. All multiprocessor programs also must be painfully glued together, because, though we have made strides in the automatic scheduling of multiple processors [3], we have not built that into the Toolkit code generator.

Nevertheless, despite its prototype status, the Toolkit demonstrates a means practical within the limits of current technology, to provide relatively inexpensive supercomputer performance for a limited, but important class of problems in science and engineering. The key is to avoid the generality—both in architecture and in compilation technology—that cause computers of comparable speed to be expensive to design, build, and program. The result is machines that are not as fast as the fastest supercomputers, but whose price advantage permits them to be used for applications that would be otherwise infeasible.

Acknowledgements

The Toolkit project would not have been possible without continual support and encouragement from Joel Birnbaum. Henry Wu devised the board interconnection strategy. John McGrory built a prototype host interface and debugging software, and Carl Heinzl wrote an initial set of functional diagnostics. Dan Zuras spent his vacation coding high-precision scientific subroutines. Rajeev Surati wrote a simulator that has become a regular part of our software development system. Karl Hassur and Dick Vlach did the clever mechanical design of the board and cables. Albert Chun, David Fotland, Marlin Jones, and John Shelton reviewed the hardware design. Sam Cox, Robert Grimes, and Bruce Weyler designed the PC board layout. Darlene Harrell and Rosemary Kingsley provided cheerful project coordination.

References

[1] H. Abelson, A. Berlin, J. Katzenelson, W. McAllister, G. Rozas, and G.J. Sussman, "The Supercomputer Toolkit and its Applications," *Proc. of the Fifth Jerusalem Conference on Information Technology*, Oct. 1990. Also available as AI Memo 1249.

[2] J. Applegate, M. Douglas, Y. Gürsel, P. Hunter, C. Seitz, G.J. Sussman, "A digital orrery," *IEEE Trans. on Computers*, Sept. 1985.

[3] A. Berlin, "Partial Evaluation Applied to Numerical Computation," *Proc. 1990 ACM Conference on Lisp and Functional Programming*, Nice France, June 1990.

[4] A. Berlin and D. Weise, "Compiling Scientific Code using Partial Evaluation," *IEEE Computer* December 1990.

[5] Bipolar Integrated Technology, "B2110A/B2120A TTL Floating-point chip set," Bipolar Integrated Technology, Inc., Beaverton, Oregon, 1987.

[6] IEEE Std 1178-1990, *IEEE Standard for the Scheme Programming Language,* Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[7] Integated Device Technology, *High-Performance CMOS Data Book Supplement,* Integated Device Technology, Inc., Santa Clara, California, 1989.

[8] J. Laskar, "A numerical experiment on the chaotic behaviour of the Solar System", *Nature*, vol. 338, 16 March 1989, pp. 237–238.

[9] Joel Moses, "Toward a general theory of special functions," *CACM*, 25th Anniversary Issue, August, 1972.

[10] Thomas R. Quinn, Scott Tremaine, and Martin Duncan "A Three Million Year Integration of the Earth's Orbit," *Astron. J.*, vol. 101, no. 6, June 1991, pp. 2287–2305.

[11] G. J. Sussman and J. Wisdom, "Numerical evidence that the motion of Pluto is chaotic," *Science*, Volume 241, 22 July 1988.

[12] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, *Numerical Recipes: The Art of Scientific Computing,* Cambridge Univ. Press, 1986.

[13] J. Wisdom and M. Holman, "Symplectic Maps for the N-body Problem," (to appear).