R1000 Hardware

Preface

This is an overview of the R1000 hardware. It is a document that can be used by new employees and will be used as source material to be included in other documents such as the FRU (Field Replaceable Unit) manual, or system overview manual (maybe others).

The reader for this document is a new employee to the technical support areas or manufacturing. The reader is assumed to be familiar with current minicomputers or microprocessors.

This is a company proprietary document! It will not be distributed outside of Rational!

Contents

Physical System Overview . . . 1
- Processing System . . . 2

Logical System Overview . . . 7
- Functional Organization . . . 7
  - Ada-based machine . . . 7
  - CPU design . . . 8
  - IOS design . . . 9
- Specs and Schematics . . . 9
- System clock generation . . . 11
- CPU Microarchitecture . . . 12
  - Facilities . . . 13
  - Events . . . 13
  - Control Store . . . 14
- Diagnostic System . . . 15

Memory Board design . . . 21
- Functional Design . . . 22
- Microcode design . . . 29
- Diagnostic design . . . 29

Microsequencer Board design . . . 33
- Functional Design . . . 33
- Microcode design . . . 34
- Diagnostic design . . . 34

Val Board design 35
- Microcode organization 36
- Functional Design 36
- Diagnostic design 41

Type Board design 45
- Microcode organization 46
- Functional Design 48
- Diagnostic design 52

Field Isolation Board design 55
- Microcode organization 56
- Functional Design 57
  - Rotator/Merger 57
  - Memory Monitor 62
  - CSA Monitor 64
- Diagnostic design 66

Sysbus Board design 70
- Functional Design 70
- Microcode design 71
- Diagnostic design 71

I/O Adapter Board design 72
- Functional Design 72
- Microcode design 73
- Diagnostic design 73

Examples of Operations 74

Figures

Main Rack 2
Main Chassis Layout 3
Sysbus/Processor Interconnection 4
CPU Buses 5
CPU Quarter-Cycle Clock Signals 11
Diagnostic Microprocessor System 16
Slave Microprocessor with Experiment Logic 18
Examples of Rotator/Merger Operation 60

Tables

Memory control codes . . . 24
Tag Contents . . . 26
Page State field contents . . . 27
Tag store address (hash function) . . . 27
Data Store RAM address bits . . . 28
DFSM stimuli, Memory board . . . 30
Microcode fields, Val board . . . 36
Random Field Encodings, Val board . . . 36
Example Microinstruction, Val board . . . 37
DFSM stimuli, Val board . . . 42
Microcode fields, Type board . . . 46
Random Field Encodings, Type board . . . 47
Microevents, Type board . . . 48
Example Microinstruction, Type board . . . 48
DFSM stimuli, Type board . . . 53
Microcode fields, FIU board . . . 56
Operation Select Encodings, FIU board . . . 56
Microevents, FIU board . . . 56
Example Microinstruction, FIU board . . . 58
Field control generation . . . 60
Memory Address Register . . . 63
DFSM stimuli, FIU board . . . 67

Chapter 1: Physical System Overview

The R1000 is a large multiprocessor system housed in a single double-width rack or frame. There are one to four peripheral (19") racks (sometimes called mass storage racks) on the right of the main rack. There may also be peripheral racks on the left of the main rack. Terminals and printers also are included in the system.

The main rack contains one to four central processor units (CPUs) and one to four I/O subsystems (IOSs). There are also 32 or 64 asynchronous communication lines and connectors. A remote diagnosis auto-dial modem, connections for the system console, power supplies and ventilation fans are also in this rack.

The right hand peripheral racks contain all of the mass storage in the system (disk and tape units). Each rack contains a 6250 BPI streaming tape unit and may contain four 475 Mbyte disk units.

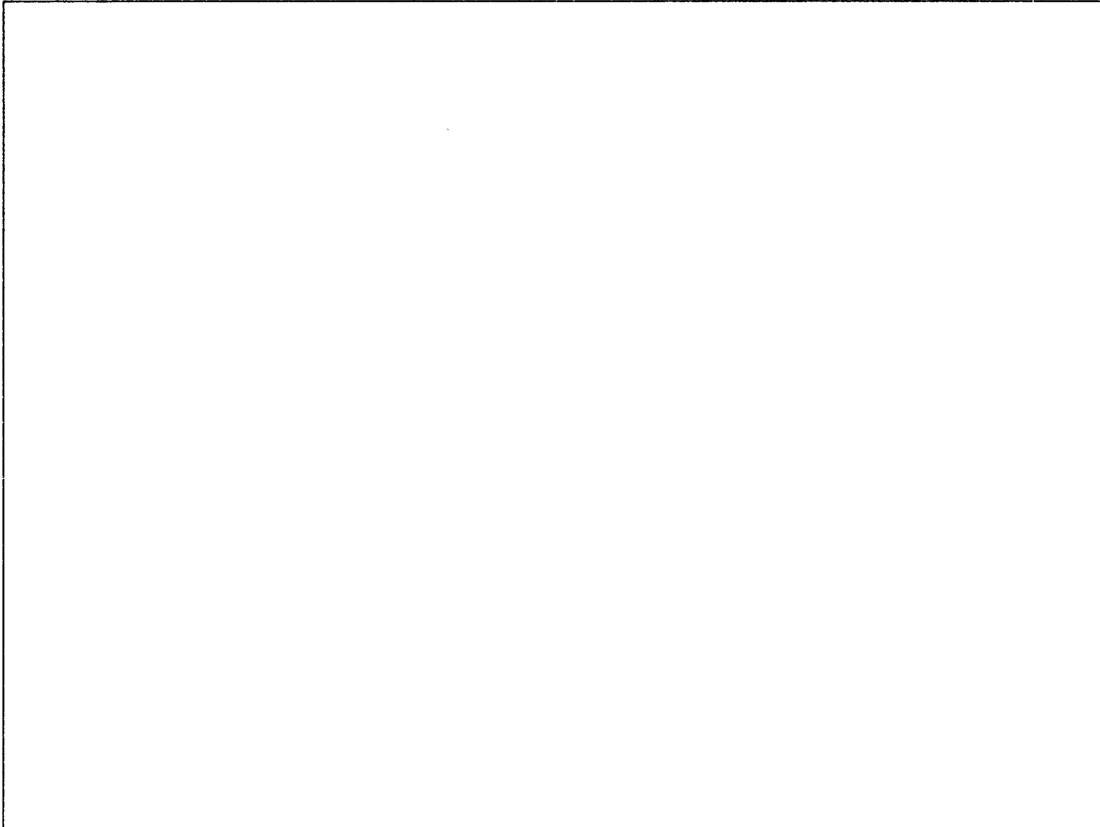
The left hand peripheral racks contain additional communications lines, network lines, and modems. Up to 192 additional asynchronous communications lines can be added in these racks (64 to a rack), beyond the 64 lines that can be put in the main rack. Ethernet and X.25 network connections can be added in these racks. Modems for the asynchronous lines can also be added.

Terminals are 64 line by 80 character in size. They are supplied by C. Itoh and are called CIT-500. Other terminals, like VT-100s, may also work and may be allowed on our system. The CIT-500s are what we will sell.

Printers will initially be a line printer. There are several we could use and we have not chosen yet. Later, we may add a laser printer for better quality output.

Figure 1–1

Main Rack



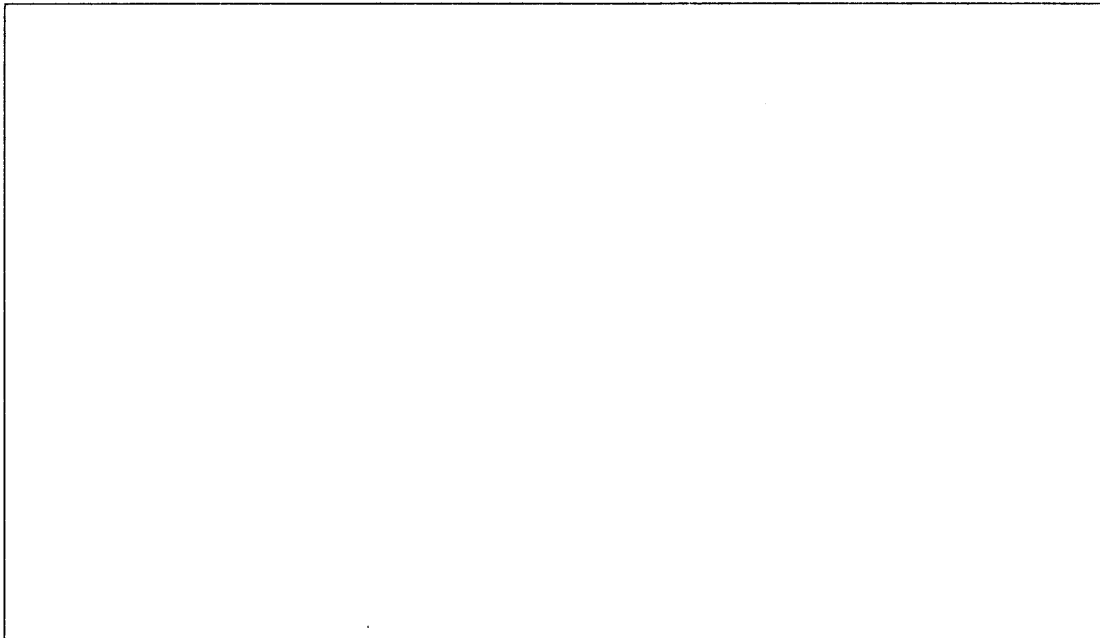
Processing System

The processing system in the main rack is the up to four CPUs and up to four IOSs. The logic for the system occupies one large chassis of vertical PC boards plus up to four PDP11/24 computers in their own smaller chassis below the main chassis. Behind the PDP11/24s are the power supplies and ventilation fans for the system. All these components are shown in Figure 1–1.

Each CPU consists of 9 large PC boards. They are 19.5 inches by 21 inches and contain 600-700 chips apiece. These PC boards are the:

- Memory boards (4 of them - 713 chips apiece)
- Microsequencer board (659 chips)
- Field Isolation board (665 chips)

Figure 1–2
Main Chassis Layout



- Value (henceforth called Val) board (603 chips)
- Type board (589 chips)
- Sysbus board (593 chips)

Each IOS consists of 1 large PC board (the I/O Adapter or IOA) and a PDP11/24 (sometimes called the IOP). The IOA, like the CPU boards, is 19.5 inches by 21 inches and contains roughly 650 chips. The IOA connects the PDP11/24 to the other processing elements in the system. The PDP11/24 is connected to the IOA via the UNIBUS (DEC's standard I/O bus).

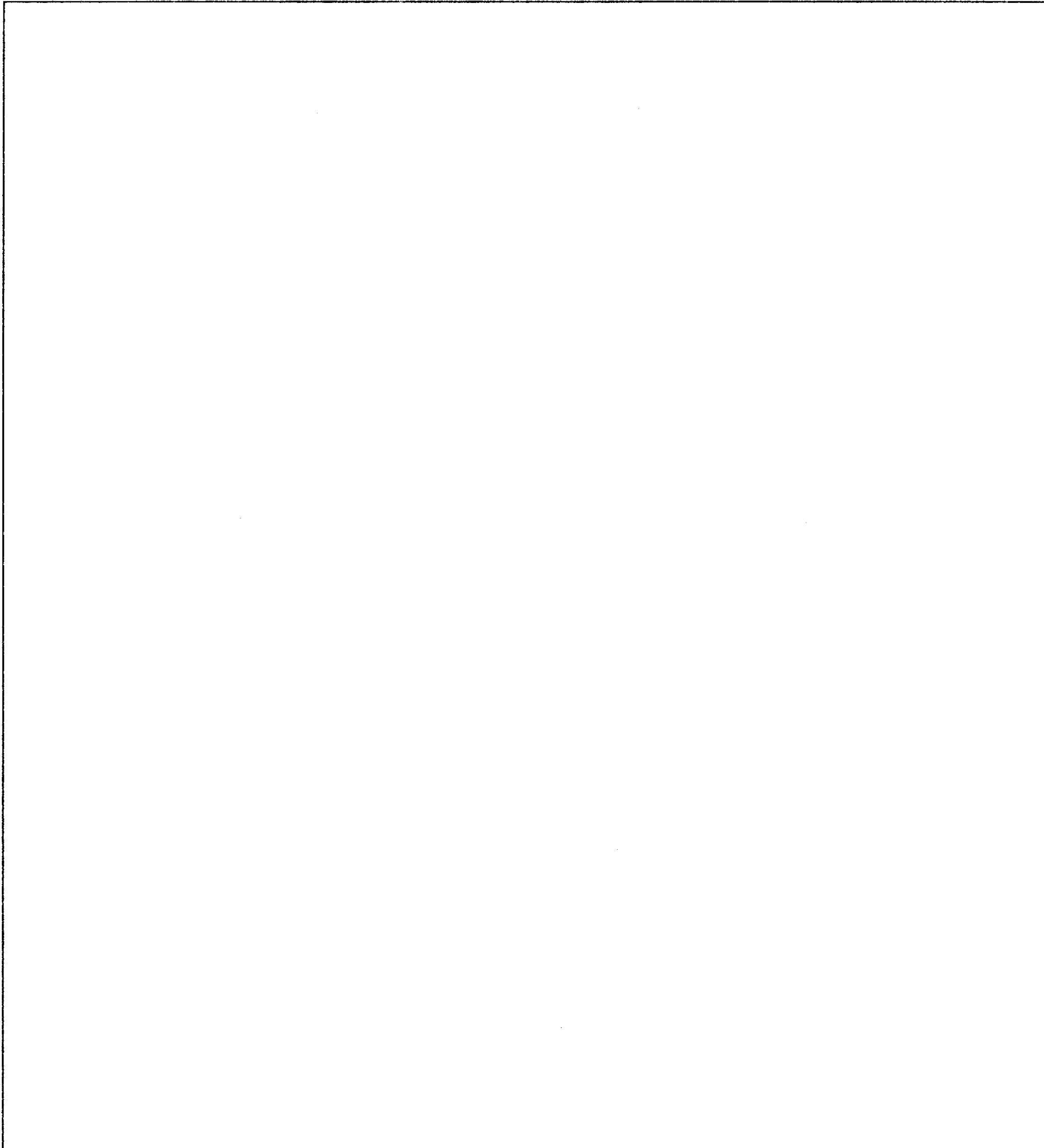
These boards and where they reside in the main chassis are shown in Figure 1–2.

The four CPUs and four IOAs are connected via a single 64-bit bus known as the Sysbus. This bus allows any of the CPUs or IOAs to communicate with any other CPU or IOA. This can be done at the rate of 40 Mbytes/second.

Figure 1–3 shows the four CPUs and four IOAs and how they are connected via the Sysbus.

Figure 1–3

Sysbus/Processor Interconnection



The 9 boards of each CPU are logically connected via a set of buses. These buses are the:

- Type bus - 64 bits
- Value (henceforth called Val) bus - 64 bits
- FIU bus - 64 bits

- Address bus - 67 bits
- Microaddress bus - 14 bits

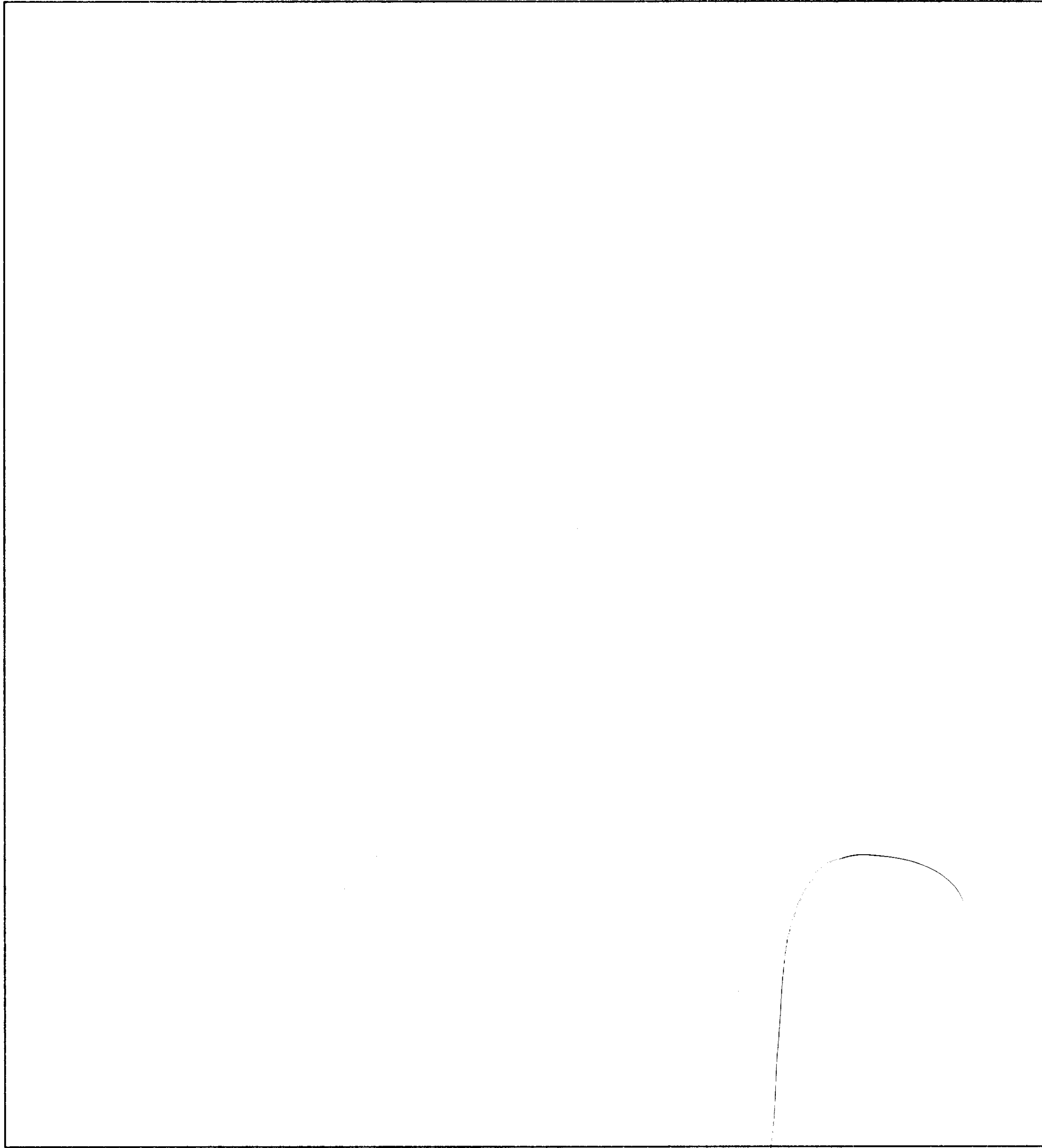
These buses connects the nine boards in each processor over a foreplane and a backplane. There is a separate foreplane for each CPU and a single backplane for the entire system. The foreplane carries the Val, Type, and FIU buses while the backplane carries the Address, Sysbus, and Microaddress buses.

All of these buses also have parity error checking (usually byte parity). The parity checking is sometimes called "odd parity". However, that becomes confusing when the bus is sometimes inverted and sometimes not. The easiest way to remember the parity algorithm is: if all of the bits (including the parity bit) are a 1, then there is a parity error.

Figure 1–4 shows the nine boards of the CPU connected via that set of buses.

Figure 1–4

CPU Buses



Chapter 2: Logical System Overview

We will look at each board individually but first we must look at some of the system-wide logic designs that transcend individual boards. We will also look at the schematics and specifications that are available.

Functional Organization

The R1000 is a multiprocessor system containing from 1 to 4 processors. Each processor is identical (except perhaps in its quantity of memory). Each processor is microprogrammed having its own control store and sequencing logic.

The basic architecture of the processors has many points:

- stack machine
- segmented memory
- virtual memory
- strong typing
- support complex data structures like discriminated records and arrays with lots of dimensions

The basis of this architecture is that the R1000 was designed to run Ada, the new high-level language from the Department of Defense.

Ada-based machine

The R1000's main function is to run Ada programs. Ada is a very extensive high-level language that puts new and unique demands on a system.[1]

Ada is a highly typed language which means that each variable (in Ada, variables are called objects) has additional information with it besides its value. When we think of an integer, we think only of its value. But it also has limits (in most cases the limits of the machine to represent integers). These limits are part of the object's type.

Ada lets you define objects with arbitrary limits. For example, you can define an object that only has "legal" values of 0 to 10. The machine (in our case) is responsible for making sure that the object does not get a value higher than 10 or lower than 0.

Likewise, Ada does not let you store floating point numbers in integer objects and visa versa. The architecture must prevent this from happening.

Some other machines claim to deal with this strong typing. In general they accomplish it by running extra code within the programs to check for limits and types. This becomes very inefficient compared to our machine.

CPU design

This strong typing is just one example of how the language Ada forced the CPU (hardware and microcode) to do some not-so-obvious things. It also explains why there are separate boards for the type information (the Type board) and for the data or value information (the Val board).

Something that Ada did not force is the mechanism for the bit-packing of arrays and records. Imagine having a large array of boolean values. In many current machines, each boolean is forced into a separate memory location: very expensive when each memory location is 128 bits like ours are.

So the CPU allows bits of an array to be packed into the smallest space they would otherwise occupy. But this requires hardware to pack and unpack quickly. Otherwise the gain in memory compaction is lost in extra processing to pack and unpack. This hardware is the Field Isolation Unit or the FIU board.

[1]For more information on Ada, see Grady Booch's book *Software Engineering with Ada*; there are copies available here.

Two other boards in the CPU are fairly obvious what they do. One is the memory board(s). It stores 2 Mbytes organized as 128K locations of 128 bits per location.

The other obvious board is the microsequencer board. It provides the sequencing through microcode, generating microaddresses, decoding instructions, and responding to various types of interrupts (called "events").

The last CPU board is the Sysbus board. It is responsible for communicating with the other CPUs and IOSs in the system. They are all connected via the Sysbus and this board provides the buffers and protocol handling for that bus.

IOS design

The remaining logic components of the system make up the I/O System(s). Its design is not governed by Ada like the CPU is. Instead it was governed by the realization that it is impractical to design an entire new I/O system including all the peripherals and their controllers.

So a common, readily available I/O bus, I/O processor and peripherals were chosen. The only logic that had to be designed then was an adaptor board that connected the I/O processor with the Sysbus.

This I/O processor (IOP) is Digital Equipment Corporation's PDP11/24 and the I/O bus is the Unibus. The IOP controls its I/O devices via the Unibus. We can attach peripherals from a large collection that are made for the Unibus.

The adaptor board is called the IOA board. It provides functions similiar to the Sysbus board of the CPU (buffers and protocol handling for the Sysbus). The IOA also provides the system console control via the "master" diagnostic microprocessor it contains. (The diagnostic microprocessors will be discussed further later in this chapter).

Specs and Schematics

There are specs for each board. Their principle function is to provide the spec for the microcode that exists on the board. These specs were used to define the microcode simulator. They are of use only for finding the definitions of the

microcode for each board. They don't describe much about how a board does any particular function. And there isn't a spec for the memory board, though there is a spec on the memory monitor functions (generates controls for memory) which are distributed over several boards.

The schematics are produced for each board on the Daisy logic capture systems. They include a set of block diagrams. However these block diagrams are suboptimal (a cute but overused word here that means "less than great", often "much less than OK").

These block diagrams have several restrictions placed on them by the Daisy system that makes them that way: each block *must* have every input and output on it that the corresponding page(s) of schematics do; each block *must* correspond to one or more full pages of schematics; and blocks can not be created that constitute less than one full page of schematic.

This leads one to believe that the block diagrams are worthless. Not quite. Many large blocks are represented well in the block diagrams. It is the subtle blocks that don't easily correspond to pages of the schematic that are a problem: they don't appear on the block diagrams at all. Nonetheless, we must use these block diagrams for our discussions since you probably will be using them and the schematics at some time.

The schematics have several conventions worth mentioning. One is the use of '~ ' (tilda). It usually appears as the last character of a signal name. It represents inversion (SIGNAL~ represents the inverse of SIGNAL).

Also the use of signal name extensions (like filename extensions) to signify multiple copies of the same signal. If, for example, SIGNAL was required to drive so many loads that one signal was not enough, you would see SIGNAL.B0 and SIGNAL.B1 as two separately-buffered but otherwise equivalent signals. Functionally, those two signals are identical.

Another example of signal name extensions is for board prefixes. The two signals T.MUMBLE and V.MUMBLE are identical except that the first appears on the Type board and the second appears on the Val board.

The bit numbering is sometimes a source of confusion: the industry is not consistent which bit in an arbitrary word is bit 0. We number bits from left to right: bit 0 is the left-most or most-significant bit.

A memory "word" also needs to be defined. Our memory boards always transfer 128 bits at a time. While some of those bits may be ignored or shifted around by the processing logic, and there is no 128-bit bus, and the meaning of those 128 bits is very context sensitive, we will from time to time use the word "word" when refering to the 128 bits from memory.

It is also worth mentioning while we are talking about memory words that they are usually divided into two separate halves: the Type half and the Val (Value) half. The division is so common that there is not a single memory data bus but two buses, the Type bus and the Val bus, that transfer data to or from memory.

System clock generation

All systems must have a set of clocks for synchronizing data transfers and functions within the system. Our clock scheme begins on the "master" IOA board, IOA0. There a master oscillator generates a 40 Mhz clock CLK.8X. This is divided into two other clocks which are distributed to all the Sysbus boards and IOA boards in the system. They are a 20 Mhz clock, PROC0.CLK.4X, and a 5 Mhz phase clock, PROC0.PHASE.

Each Sysbus or IOA board takes the two clocks and produces two other clocks, CLK.2X.BP (both positive and negative versions), and PHASE.BP. These clocks are distributed within each processor.

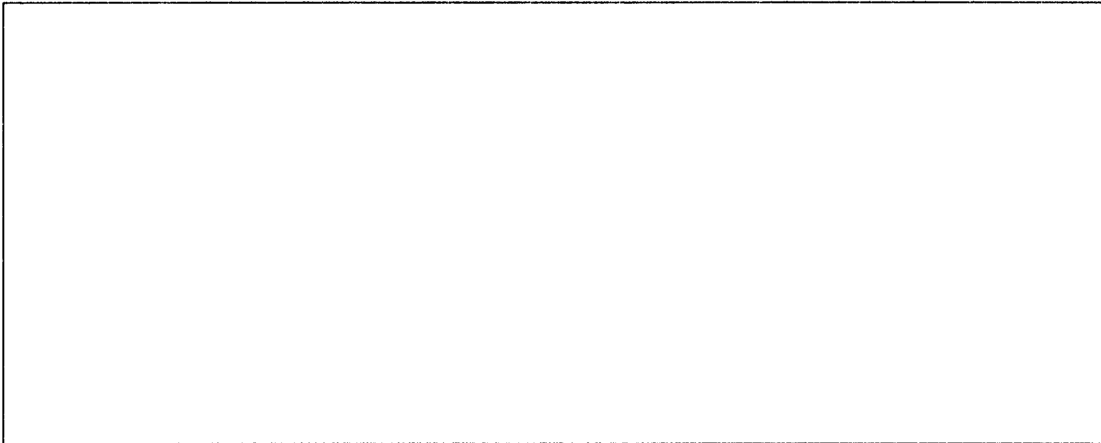
Each processor board takes these two clock signals and produces the necessary clock signals for the board. In general, that means producing four "quarter" clocks, Q1~ through Q4~. Each quarter clock is a 75% duty cycle clock that are each 90 degrees out of phase from each other. In other words, each produces a 50 ns. pulse and each pulse does not overlap with any other pulse. Each microcycle is composed of four quarters. The microcycle begins and ends between Q4~ and Q1~. Figure 2–1 shows these signals.

Another pair of signals that many boards use is H1 and H2. These signify the first and second half of a microcycle.

Most boards however, use a clock signal with the .SCLK extension for clocking microcycles. This is because a microcycle can be aborted for several reasons.

Figure 2–1

CPU Quarter-Cycle Clock Signals



The figure shows the four signals the make up the primary data movement clocks in the system. The boundary between Q4 and Q1 marks the end of one microcycle and the beginning of another.

When this happens the "state clocks" are held or stopped to prevent the results of a microcycle from being latched. Thus clocks with the **.SCLK** (or similar) extension are not free running and may be stopped by various abort mechanisms.

There are two mechanisms for stopping these state clocks. One is for a parity error detected in the processor logic. This would result incorrect results. Each board can "freeze" the entire processor when such an error occurs with the signal **FREEZE**. This signal, when active, prevents the processor from continuing until the diagnostic system reactivates it.

The other mechanism for stopping the state clocks is used much more frequently and with much less disastorous results. At certain times in the normal operation of the machine, the sequencing logic assumes certain events won't happen. When they do, the sequencing logic forces the processor to wait a cycle before continuing processing. These events include all microevents, some macroevents, and bad hints explained in the microsequencing chapter). The board generating the event causes the processor to stop with the signal **STOP**.

CPU Microarchitecture

The microcode implements the instruction set of the machine using the hardware. In the R1000, this instruction set is very complex as suggested before. The microcode is correspondingly complex. The microcode is very wide; over 200 bits are used in each machine cycle.

Facilities

The hardware provides many features for the microcode to use in implementing the instruction set:

- 8 Mbyte "associative" memory - supporting the multi-segmented virtual address space
- ERCC memory - corrects single bit errors, detects multiple bit errors
- two 64-bit ALUs - allow type and value operations concurrently
- hardware accelerators for control stack - provides register-to-register speed on a stack machine
- 64-bit shifter/field extractor - minimizes overhead for bit-packed data structures such as arrays and records
- internal parity checking on all data paths and static RAMs
- hardware 16-bit by 16-bit multiply
- dedicated hardware to support type checking
- dedicated hardware for addressing multiple frames in stack

These facilities will be discussed further as each board's capabilities are discussed in the following chapters.

Events

The microcode responds to "events". Events are similiar to interrupts but can be generated by many different sources. There are two types of events: macro and micro. Both macroevents and microevents will cause the flow of control to change temporarily.

Macroevents are events that can only occur on between (or at the beginning of) instructions. Therefore, there is no microcode state to be saved. Instruction execution can begin again after the event is handled easily.

Macroevents include:

- Instruction queue empty
- Interrupts from other processors
- Memory refresh
- Memory reference resolution
- Control Stack accelerator overflow or underflow
- Top-of-stack address resolution

Microevents are events that can occur during any microcycle. These events, while not necessarily more severe, require the entire microcode state to be saved so that processing can continue from the exact microcycle that caused the event.

Microevents include:

- Page crossing
- Page fault
- ERCC correctable error
- Class check
- Privacy check

Events can be generated by any board and are funnelled to the microsequencer for dispatch.

Control Store

The control store for the microcode is distributed. In many machines, the control store is on the microsequencer board. In the R1000, the control store is distributed on each CPU board (except memory boards).

There are several advantages to this. Each board can run its own microcode independently (without the other boards). This means each board can be "brought up" stand alone. It also means the diagnostic programs can exercise the board very thoroughly via on-board control store. It also means that the microsequencer board does not have to send out 200 separate signals (the microinstruction); it only sends out the address of the microinstruction (14 bits.).

Diagnostic System

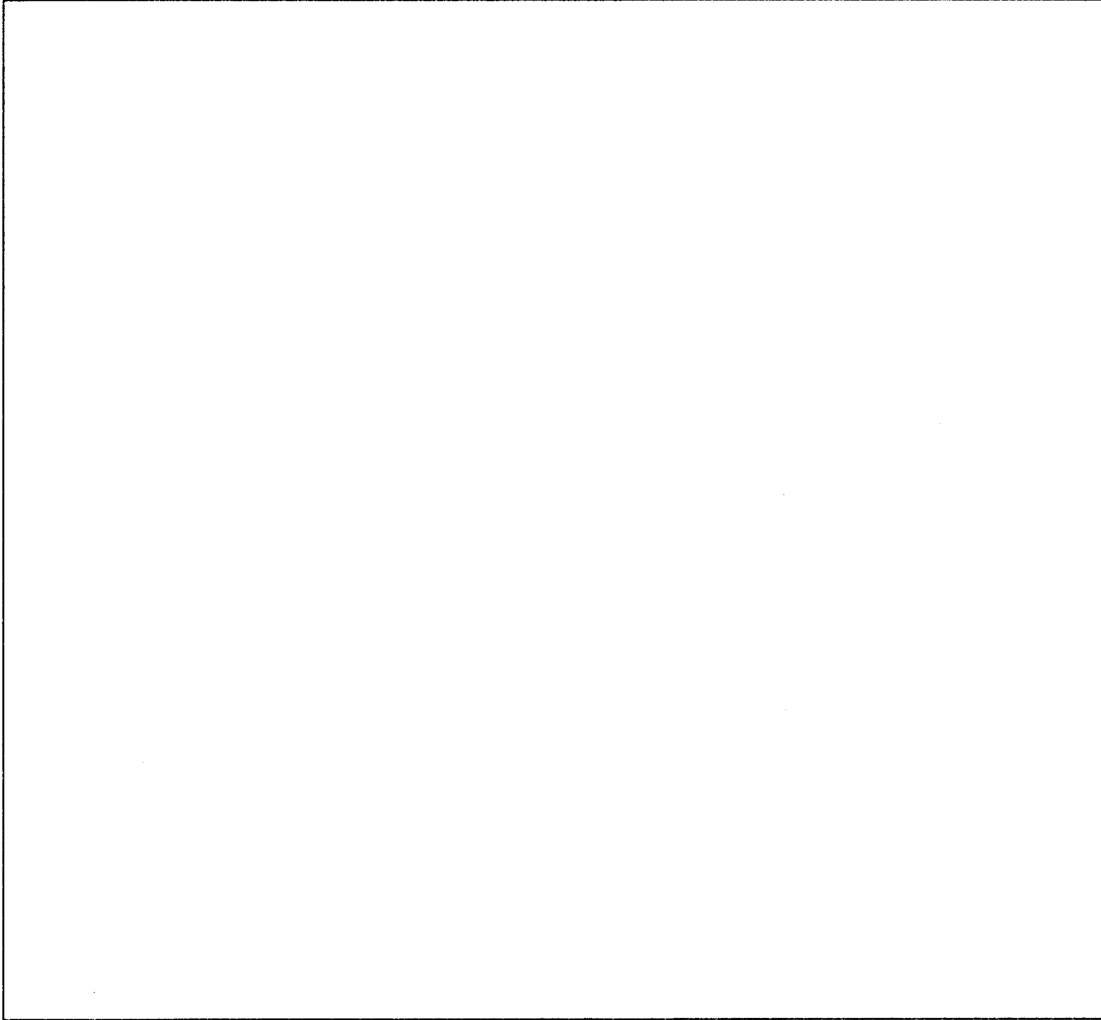
The diagnostic system is both the control of the master console (system console) and the assurance that the entire R1000 system is properly operating. It performs many tasks such as:

- Performs the power-up sequences:
  - Resets all system elements
  - Insures power is within tolerances
  - Performs power-up tests on all system elements
  - Loads diagnostic microcode into all system elements
  - Runs short diagnostic tests on all system elements
  - Loads system microcode
  - Loads initial program image
  - Begins running the programming environment
- Monitors all system functions for "fatal" errors
  - Records all system errors
  - Shuts down offending processor
  - Calls Rational service center to notify us of error(if allowed)
  - Reconfigures system to run without offending processor (if possible)
- Runs diagnostic programs when problem is suspected
  - Can isolate problems to FRU or sub-FRU level
  - Provides support for component-level troubleshooting including scope loops, logic analyzer triggers, etc.
  - Can be run remotely (via remote modem connection to Rational service center)
  - Can be run automatically on any system element whenever it detects a "fatal" error
- Runs system console
  - Provides access to system error logs
  - Provides debugging aids

The diagnostic system is composed of a set of microprocessors. In a sense, there is a master microprocessor on the IOA board which resides in slot xx (called IOA0).

Figure 2–2

Diagnostic Microprocessor System



The figure shows the set of slave microprocessors (one per board) and how they are connected to the master microprocessor on IOA0. Polling is used by the master to determine what conditions exist in the slaves.

All other CPU and IOA boards contain "slave" microprocessors that provide rudimentary diagnostic capability and error detection. The master microprocessor polls the slaves for information or downloads them with diagnostic instructions. All of this communication is done over serial asynchronous comm. lines that are built into the backplane of the R1000. Figure 2–2 shows these microprocessors and their connections to the various boards in the system.

Operation of the slave microprocessors is based on the notion of experiments. Experiments are a sequence of primitive operations and data manipulations that are run on the slave microprocessors (8051s). The data manipulations are instructions and subroutines on the slaves. The primitive operations are sequences of commands sent to various components on the particular board. These primitive operations are called DFSM commands (pronounced Dee'-fi-sum for Diagnostic Finite State Machine) and they are different on each board.

DFSM commands are an important mechanism. Each board runs at 200 ns. per microcycle when executing Ada programs, yet the 8051s could not keep up with that speed. The DFSM commands (and the corresponding hardware for implementing them) provides the synchronization mechanism for enabling the slower 8051 to, in a sense, single step the board. A DFSM command is executed by the board at full speed.

Each slave contains a set of up to 32 DFSM commands it can run on its board. They are stored in PROMs. Each DFSM command can also have up to 8 modes or qualifiers.

Each command may be a series of steps that the board must go through. There can be up to 16 steps in each command. This is reflected in the DFSM state bits.

When a slave is told to run a certain experiment, it sends out the appropriate data on its command bus for each DFSM command in the experiment. When each command is complete, the slave is interrupted and the results of the command (if any) are available on the slave's data bus.

The sequencing of steps within a DFSM command is accomplished with a PROM and register pair (the DFSM). The DFSM outputs feed one or more pairs of PROMs and registers which generate stimuli - signals that control or effect the operation of the board. These stimuli do things like:

- Load and store registers
- Increment counters
- Enable and disable drivers
- Force errors (like parity errors)
- Control board states
- Initialize board registers

With these stimuli, it is possible for DFSMs to perform, in general, any function

the board is capable of performing, in a controlled and isolated manner, such that any problems with the board are quickly discovered.

One additional function that is performed with these stimuli is the reading and writing of the non-volatile RAM or "NOVRAM". This RAM contains things like the board ECO level and other semi-permanent information about the board. This type of RAM can retain its information without power. So when the board is powered on or reset, it can find out about itself by reading this RAM. This is important for maintaining correspondence between the microcode and hardware, and for tracking by customer service and manufacturing.

Figure 2–3 shows the typical logic for the slave microprocessor, the DFSM and a set of stimuli PROMs.

The slave microprocessors often need to read and write large registers over its 16-bit bus. This includes state registers that must be initialized or interrogated after an error. Most DFSM commands require one or more registers on a board to be read or written. The slave accomplishes this with one or more **scan chains**.

Scan chains are a serial linking of registers such that the registers can be loaded and used in parallel during normal operations or can be serially read or written into the slave. This allows the slave to shift arbitrary data into or out of any register of any size on a board.

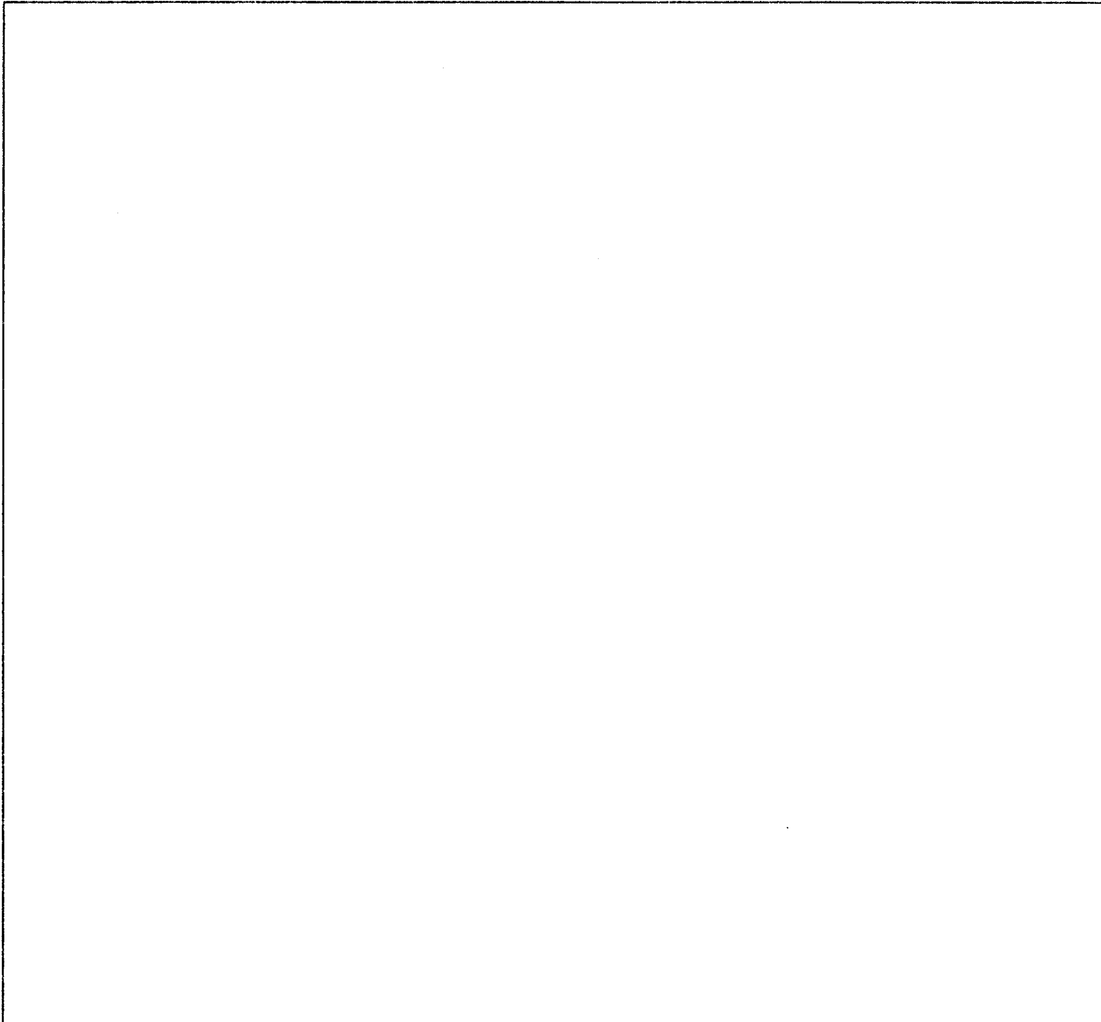
Scan chains usually link multiple registers together. However, there may be more than one scan chain on a board. Scan chains will be discussed for each board.

Often, these scan chains bring data into the slave in a very disjointed fashion. The scan chains may bring a byte at a time into the slave which then has to rearrange the bits into the correct order. This is called "permuting the bits" and the slave has code specifically to permute its scan chains into complete registers. In this fashion, reading or writing a typical register requires a number of scan chain shifts and then a subroutine in the slave to arrange the bits properly.

One last point about the slave microprocessors. Typically, they are passive during the actual running of the R1000. That means they sit there quietly waiting for some error to happen or some command to come from the master microprocessor: they don't participate in the running of R1000 code.

Figure 2–3

Slave Microprocessor with Experiment Logic



The figure shows the typical DFSM (Diagnostic Finite State Machine) logic that implements a set of stimuli-producing commands on each board. These commands are used by the slave microprocessors to run experiments on a board to verify its proper operation.

There are cases however when that is not true. The DFSM stimuli have the capability of changing state at every half cycle. This capability is used in some cases to make some stimuli active signals even when the slaves are passive. This can be a confusing attribute of the slaves in a circumstance that would imply that all stimuli are unchanging.

We'll point out such signals when they occur.

Chapter 3: Memory Board design

The memory board has several functions:

- provide 2 Mbytes of storage
- computes the logical to physical address algorithm for those pages currently on this board.
- maintain a set of "least recently used" locations for optimal management of page replacement

This really is the set of things that the memory board does. But that isn't necessarily "how" the memory board does them. The following list is the set of functional blocks that make up the memory board. They are also shown on the block diagrams.

- two data stores of 137 64K by 1 RAMs each (plane 0 and 1)
- two tag stores of 18 1K by 4 RAMs each (tag store 0 and 1)
- a 145-bit write data register (128 data, 9 check, 8 parity)
- two 137-bit read data register (128 data, 9 check)
- two 72-bit tag value registers (64 data, 8 parity)
- a 60-bit address register
- address generation/multiplexing logic
- RAM control logic
- general control logic
- diagnostic processor

That is the basic blocks you'll see on pages 1 and 2 of the schematics. But that isn't the "how" either. Unfortunately, we have to use words to describe how the board does the first list using the second list.

Functional Design

The memory board contains 2 Mbytes of storage. This is two planes of 137 64K RAMs. The 137 bits per location is divided into 64 bits for the TYPE bus, 64 bits for the VAL bus, and 9 check bits.

The memory board at first glance looks ordinary enough. The large RAM array, read and write data registers, address register and multiplexing all are very common components of any memory board. The complexity of the board first becomes apparent when you start looking closer at the address logic. Why is there a 60-bit address register when there is only 2 Mbytes of storage onboard? Why is there those tag stores and tag value registers? And how does the board know what locations are "least recently used"?

We'll try to answer those questions and more.

The starting point for the discussion is that the address is a "logical" address. The translation from a 60-bit logical address to a 17-bit physical address is a very obscure process on each memory board. This is where the tag store comes in.

The 2 Mbytes of storage are broken up into 1 Kbyte "page frames". Each page frame can contain 64 128-bit words (or locations). Each page frame has associated with it a tag. This tag contains the logical address of the physical page that is currently in the page frame. There are millions of logical pages (actually 2^{45} pages or about 40 trillion) that can be put in each page frame: the tag tells you which one really is in there.

Now, we can show how the 60-bit address is dealt with. The address is broken into the following pieces:

- 8 bits of VPID (virtual processor ID)
- 24 bits of segment name
- 3 bits of address space
- 19 bits of page select
- and 6 bits of word offset

To be fair, we should point out that there are 7 additional address bits (a total of 67) but the least significant 7 select a bit within a word. That bit selection process takes place on the FIU board so are not needed on the memory boards.

When an address is latched into the address register at the end of a microcycle (pages 5 and 6 of schematics) it is broken into those parts listed above. Some of the segment, space, and page select bits are sent through a "hash function" (page 14) to select a location corresponding to a page frame in each tag store. (We'll come back to this hash function.)

The selected tag store locations are interrogated (page 16 and 19) during Q2~ and Q4~ of the following cycle. The output of the tag store and the supplied logical address are compared to check (page 17 and 20) generating NAME MATCH and PAGE MATCH. If both are true for either tag store during either Q2~ or Q4~, then there is a hit (page 25) on the corresponding plane.

A hit means the selected page frame contains the requested logical page. This allows the main RAMs to be addressed with the same address that addressed the tag store along with the word select bits (pages 14 and 30 - 39). The read data then is supplied via the read data register (pages 11 - 13) or the write data is supplied with the write data register (pages 8 - 10). This all happens during the cycle following the tag store interrogation (memory cycle 2).

If a hit does not occur on this board (or any other memory board), a microcode event happens (an interrupt to the microsequencer) and the microcode resolves the "page fault".

That is how a basic memory cycle works. You'll want to get that basic cycle thoroughly down before plunging in much further (it gets a lot worse).

Why are there two tag stores; wouldn't one be enough? And why are the tag stores interrogated twice (in Q2~ and Q4~)?

The answer lies in performance reasons. You want as many interrogations per memory request as possible. That increases the likelyhood that one of the interrogations will produce a hit. (A hit is much higher performance than the alternative, a miss, which results in accessing a disk.) So each memory board provides 4 "sets" of tags and each set is checked for each memory request. There are 512 "lines" of sets.

The memory provides this with two banks of tag store which it then time-multiplexes each into two halves. That makes it appear that there are 4 sets of tags. Two sets (one from each physical set of RAMs) are checked in Q2~ of the first memory cycle and the other two sets are checked in Q4~ of the first memory cycle.

But can't more than one of these sets produce a hit? What if there is a hit on more than one memory board: won't that mess things up?

Well, yes it would if the hardware and microcode didn't cooperate to prevent it. They must make sure that each of the 16 sets (4 on each of 4 boards) contains a different tag for each page frame.

They do this by careful maintenance of the tags during a "miss", also called a "page fault". The microcode is interrupted and begins executing a "page fault handler". The function of this routine is to find a place for the page that missed and loaded into memory from disk. It must find this place among the 16 tags where that particular logical address can be placed.

This routine looks through the set of 16 tags that the faulted page could be put into. Its algorithm for finding an page frame is as follows:

(1) Look for any frames that are empty or available.

The new page can be placed directly into the page frame.

(2) If none exist, look for the page frame that has not been used for the longest time (least recently used) that has not been modified (has not been written to).

In this case, the new page can be written over the old page in the page frame (the old page is still on disk and does not need updating).

(3) If none exist, look for the page frame that has not been used for the longest time and has been modified.

In this case, the old page must be written back to disk before the new page can be loaded into the page frame. Note this is the most expensive option in terms of performance because of the additional disk traffic.

To support this microcode activity, the memory must provide additional access to the tags. Diagnostic operations on the memory board must also be supported. We must now discuss the total set of operations the memory board is capable of.

There are actually 16 different operations or modes of the memory. Table 3-1 lists all 16 along with the encodings on the **MEM.MODE0** through **MEM.MODE3** signals.

Table 3–1

Memory control codes

Operation	Code	LRU	Mod	Explanation
Physical Write	0	pass	pass	Bypass associative mechanisms, use VPID bits 0 - 3 to select a set, and write directly into the selected memory location
Physical Read	1	pass	pass	Bypass associative mechanisms, use VPID bits 0 - 3 to select a set, and read directly from the selected memory location
Logical Write	2	update	set	If hit, write into selected memory location
Logical Read	3	update	pass	If hit, read from selected memory location
Copy to Plane 0	4	pass	pass	Diagnostics only
Memory to Tag	5	†	†	Diagnostics only
Copy to Plane 1	6	pass	pass	Diagnostics only
Test TVR	7	†	†	Diagnostics only
Tag Write	8	write	write	Update the tag store with information about a new page
Tag Read	9	pass	pass	Read the tag store
Initialize MRU	A	pass	pass	Reset all LRUs in the tags to be equal tag's set number
Tag Query	B	pass	pass	Check if requested tag is in one of the four tags selected by the current address lines
Name Query	C	pass	pass	Check if requested name is in one of the four tags selected by the current address lines
LRU Query	D	update	pass	Check if one of the four tags currently selected by the address lines is the least recently used
Available Query	E	pass	pass	Check if one of the four tags currently selected by the address lines is available (i.e., invalid)
Idle	F	hold	hold	Do no memory operation

† During diagnostic operations, Tag Store 1 is used to save the read data. Therefore, the LRU and modified bit fields of Tag Store 1 are written with the corresponding data.

This mode is sent along with the address (which note isn't always logical). (See schematic page 24). The address and mode are sent from the memory monitor every Q4~.

We should also show exactly what is the contents of the tag store and how is it addressed (i.e., what is that "hash function").

Table 3–2

Tag Contents

Field Name	Bits	Explanation
Name Tag	0–31	The segment name which contains the selected page
Page Tag	32–50	The page within the segment
Modified bit	51	The bit specifies whether the page is dirty or clean (has been written to or not)
LRU position	52–55	The relative position of this tag location between least recently used ('0000') and most recently used ('1111')
Page State	56–57	The state of the page (defined below)
Manager Flags	58–60	Only bit 60 currently used (defined below)
Space Tag	61–63	The type of segment

Each tag location must contain enough information to provide the entire logical page address for the tag interrogations as well as status for the page fault handler. Table 3–2 shows the contents of a tag location:

The LRU bits of the tag store are particularly important. Assuming a four board system, each of the 16 sets must have status information for the page fault handler that tells it which set was "most recently used." This is the purpose of the LRU bits.

Each set's LRU bits are initialized to their static position within the 16 sets. From then on, whenever a set has a hit, its LRU bits are set to F_{16} (or B_{16} for a three board memory system, 7 for a two board system, 3 for a one board system). In addition, all sets between F_{16} and the previous LRU value of the set that had the hit are decremented. In this way, all sets have a unique LRU value and the least recently used set will have a LRU value of 0.

An example of maintaining the LRU bits might help. Let's assume a one board system for simplicity. Of the four sets on the board, we'll assume they have just been initialized. That means set 0 has LRU value 0, set 1 is a 1, set 2 is a 2, and set 3 is a 3. Now let's assume the first logical memory operation is a hit on set 1. The following steps occur.

(1) Set 1 broadcasts the fact is has a hit. It sends its set number (1) to all other boards in the memory system so they know that here was a hit and which set had it.

Table 3–3

Page State field contents

Bits	Name	Explanation
00	Invalid	All logical queries will miss
01	Read/Write	All logical operations are allowed
10	Read Only	All logical write operations will miss
11	Loading	Reserved for during a disk transfer into the page: all logical write operations will miss; logical read operations are intercepted at the FIU board and will also miss

(2) Set 1 broadcasts its current LRU number. We'll call this the previous LRU number. In the example this is a value of 1.

(3) Set 1 sets its new LRU number to 3. Remember this example is for a one board system. Thus set 1 becomes the most recently used.

(4) All sets whose LRU number is between 3 (the top most LRU value) and the previous LRU number (broadcast by the set which had the hit – in this case a 1) decrement their LRU number. Thus set 0 still has an LRU value of 0, set 1 has 3 (the most recently used), set 2 has 1, and set 3 has 2.

Notice this algorithm maintains a unique LRU number in each set.

The Page State bits of the Tag Store provide access protection for the operating system. All logical reads and writes to any particular page of memory must be explicitly allowed by the operating system when a new logical page is loaded into the page frame. Both separate read and write protection is provided.

Table 3–3 defines the Page State field of the Tag store:

Bit 60 of the tag store is the Write Protect Flag. It is used only when the page frame is in the Loading state. The loading state is when the operating system has requested a new logical page and the disk is in hte process of transfering that page to memory. The Write Protect Flag indicates whether the new page is write protected or not (it can't be write protected until after the disk transfer is complete).

Table 3–4 defines the hash function used to address the tag store:

That completes the data for the tag stores. There is one other area that has not been covered: the Data Store.

Table 3–4

Tag store address (hash function)

Address Line	Function
Line0	Segment15 xor Page18
Line1	Segment16 xor Page17
Line2	Segment17 xor Page16
Line3	Segment18 xor Page15
Line4	Segment19 xor Page14
Line5	Segment20 xor Page13
Line6	Segment21 xor Page12
Line7	Segment22 xor Space1
Line8	Segment23 xor Space2

The data store is two planes of 64K RAMs. Each plane, therefore requires 16 bits of address and a plane select.

When the tag store is interrogated (a tag query), you'll remember it uses a hash function to select which page frame the logical address may reside in. That same hash function is used for part of the RAM address.

The remainder of the RAM address is made up first of the word select bits from the logical address (remember the tag interrogations only select a page frame). These six bits are the least significant of the logical address.

One additional bit is used in the RAM address. It comes from the tag query logic and it specifies which of the two queries for that particular data plane had the hit.

Which of the two planes is selected is also accomplished with signals from the tag query logic.

64K RAMs are addressed in two parts. The first part is called a row address and is latched into the memory chips with a signal called **RAS**. The second part is the column address and it is latched into the memory with **CAS**. Table 3–5 shows the row and column addresses for the data store RAMs.

Table 3–5

Data Store RAM address bits

Address Bit	Function
Row0	Segment15 xor Page18
Row1	Segment16 xor Page17
Row2	Segment17 xor Page16
Row3	Segment18 xor Page15
Row4	Segment19 xor Page14
Row5	Segment20 xor Page13
Row6	Segment21 xor Page12
Row7	Segment22 xor Space1
Column0	Segment23 xor Space2
Column1	Set selected (when logical - otherwise VPID2)
Column2	Word0
Column3	Word1
Column4	Word2
Column5	Word3
Column6	Word4
Column7	Word5

Microcode design

There is no microcode on the memory boards. All memory functions are controlled by microcode on the other CPU boards, principly the FIU board.

There are no microcode events generated by the memory boards either. They are generated by the FIU board for things like page faults.

Diagnostic design

The diagnostic processor is on pages 21 - 23 of the schematics.

The diagnostic processor on the memory board (the slave) can read and write (directly or indirectly) all storage elements on the memory board. This is typically for verifying that the board is running correctly. The slave also can provide

status information whenever a parity error occurs in the tag store or on the Val bus or on the Address bus.

The slave can read and write the following set of functional units on the memory board:

- The 60-bit memory address register. This is accessible with 8 serial scan chains (7 are 8 bits long, 1 is 4 bits long). The bytes read or written must be permuted.
- The 8 parity bits of the memory address bus.
- The 145-bit write data register. This is accessible with 8 serial scan chains (6 are 20 bits long, 1 is 12 bits long, and 1 is 13 bits long). These bytes must be permuted.
- The 8 parity bits of tag store 0.
- The 8 RAM address bits of either data store.
- An 8-bit scan chain which includes the memory control state.
- A 7-bit scan chain which includes the LRU control state and the tag parity error bits.
- The 2 bit board ID.

The slave also has a 12-bit counter at its disposal which it can read or write, increment or clear. It can use this to address the NOVRAM on the memory board, the tag stores, or the data stores.

The slave can specify any of the memory operations discussed earlier in this chapter as well as generate any of the stimuli in Table 3–6. Between these two capabilities, the slave can perform (at full speed) any operation the memory can perform under microcode control plus several high speed operations specifically to verify the memory's correct operation.

Table 3–6

DFSM stimuli, Memory board

Signal	Function
Force Refresh	If the microcode does not refresh the memory within a certain time, the 8051 can force refresh to occur with this signal.
Scan Mar	Enables 8 bits of the MAR into the 8051.
Diag Tsadr.oe	Enables 8 bits of the Tag Store 0 value (specifically the 8 parity bits) into the 8051.
Diag Adr.oe	Forces the data store RAMs to be addressed with 8 bits from the 8051.
Diag Rd RDR	Enables the Read Data bus onto the Type, Val, and check bit busses.
Diag RTV	Enables the tag value bus register onto the Read Data bus (otherwise the data store registers are enabled).
Diag Rd TVR	Enables the Read Data bus onto the Val bus and Val parity bus (used with Diag RTV to read tag store values).
Stop RDR	Inhibits the loading of the Read Data Register (RDR).
Enable Registers	Allows the loading of the Write Data Register (WDR) and the Memory Address Register (MAR).
Freeze En	Enables the "freezing" of the CPU due to a parity error.
Diag LD MAR	Load the MAR.
Diag MAR sel	Selects parallel loading the MAR (normal case) or right shift (scan case).
MEM.CTL S0 and S1	Selects four operations on the memory control registers - load, shift left, shift right, hold.
Scan WDR	Enables 8 bits of the WDR into the 8051.
Diag SH WDR	Along with Diag LD WDR, selects loading or shifting right the WDR.
Diag LD WDR	Along with Diag SH WDR, selects loading or shifting right the WDR.
Perr S0 and S1	Selects four operations on the parity error register - load, shift left, shift right, hold.
MRU Sel0 and Sel1	Selects four operations on the memory state and hit set register - load, shift left, shift right, hold.

Table 3–6 (continued)

DFSM stimuli, Memory board

Signal	Function
Scan Control	Enables the Diagnostic address counter, board ID, and scan chain into the 8051.
Diag Ctr Ld	Loads the Diagnostic address counter.
Diag Ctr Inc	Increments the Diagnostic address counter.
Diag Xcvr.oe	Enables data to or from the 8051.
Rd Diag Bus	Directs data to or from the 8051.
Diag Buz Off	Enables data onto the Type and Val busses.
FSM Done	Signals the 8051 that the DFSM command is complete.
Diag Ctr Clr	Clears the Diagnostic address counter.
NovRAM CS	Enables the NovRAM.
NovRAM WE	Writes the NovRAM.
NovRAM Store	Puts current data into non-volatile part of NovRAM.
Bank Select	Most significant bit of DFSM state bits.
Trigger Scope	Backpanel pin for triggering scopes or logic analyzers.

Chapter 4: Microsequencer Board design

The microsequencer board has several functions:

- List'em

This really is the set of things that the microsequencer board does. But that isn't necessarily "how" the microsequencer board does them. The following list is the set of functional blocks that make up the microsequencer board. They are also shown on the block diagrams.

- List'em

That is the basic blocks you'll see on the first few pages of the schematics. But that isn't the "how" either. Unfortunately, we have to use words to describe how the board does the first list using the second list.

Functional Design

- follow simple operations of board first

- diverge into more complex operations

- refer to schematics occasionally (with page and signal name)

Microcode design

- organization of microcode fields on this board
- macro and micro events generated
- refer to spec, microsim, or microcode if appropriate

Diagnostic design

- design of microprocessor on this board (how different from others)
- set of experiments on this board
- cross reference to listings occasionally

Chapter 5: Val Board design

The Val board has one primary function: to perform all the value calculations and operations within the instruction set. This includes both integer and floating point operations.

These calculations are done in parallel to all other functions on all the other boards in the system. In particular, the Type board operations are separately controlled and done in parallel with all Val board operations. The Type board can provide range checking, privacy checks, class checks, and visibility checking in parallel with the value operations being done on the Val board.

The Val board is completely controlled by microcode. It takes a microaddress from the microsequencer, looks up the contents of its control store at that address, and executes its portion of the microinstruction.

The following list is the set of functional blocks that make up the Val board. They are also shown on the block diagrams.

- 1K by 64 bit register file
- 16-bit by 16-bit multiply logic
- General purpose 64-bit ALU
- 64-bit Val half of write data register (WDR)
- Shift-mux for selecting write data for the register file
- Leading zero counter
- 10-bit loop counter
- 40-bit control store with microinstruction register

- and the standard slave diagnostic microprocessor

The Val board also has four buses that connect the various blocks: ALU bus, A bus, B bus, and C bus.

That is the basic blocks you'll see on the first few pages of the schematics. How the Val board performs the value operations within the instruction set is really a function of microcode. We'll look at the microcode organization on the Val board briefly first. Then we'll go through the functional design of the Val board.

It should be pointed out that the Val and Type boards are very similar. The bus structure, register file organization, and many other functional blocks are the same on the two boards. The prime difference is the Val board has the multiply logic and the Type board has type checking logic.

Microcode organization

The portion of the microinstruction contained on the Val board is 40 bits wide, including a parity bit. This is broken into 10 fields. Table 5–1 provides the breakdown of these bits and a brief description of each field.

The first page of the schematic provides a summary of all encodings for all fields of the Val portion of the microinstruction. The specification for the Val board contains a full description of all the encodings of all the fields.

The Random field is particularly distinct because its operations are not associated with each other (that's why it's called random). Table 5–2 provides a summary of all the encodings of the Random field.

The Val board does not generate any macro or micro events.

Functional Design

To show how the Val board operates, we'll make up a single, simple microinstruction and show how it routes data through the board. It won't show all of

Table 5–1

Microcode fields, Val board

Field Name	Bits	Explanation
A Address	0–5	Selects the A bus value; includes sources like TOS, TOS-3, loop counter, multiplier product
B Address	6–11	Selects the B bus value; includes sources like TOS, TOS-3, Val bus, GP
C Address	12–17	Selects the C bus destination (within the register file or loop counter); includes destinations like TOS, TOS+1, GP, loop counter
Register Frame	18–22	Selects one of 32 frames within the register file for frame local A, B, or C addresses
Mux Source	23–24	Selects a (possibly shifted) source for the C bus
Random	25–28	Selects one of 16 various operations; includes incrementing or decrementing loop counter, various split bus operations, control the multiplier, count zeros
Multiplier A Source	29–30	Selects which 16 bit field is used as the A input to the multiplier
Multiplier B Source	31–32	Selects which 16 bit field is used as the B input to the multiplier
ALU Function	33–37	Selects the operation the ALU performs
C Source	38	Selects FIU bus or Mux Source for the C bus
Parity	39	This parity bit should be a 1 if all of the bits in the microword on the Val board are 1

the paths nor all of the functional units. But it will give you an easy view of how the board operates in its simplest way.

First, a description of what this microinstruction is to do. We want to read a value from memory, mask out some of the bits with a mask already in a register, and place the result into another register. The value from memory comes via the Val bus which is the least significant half of the 128-bit data to or from memory (the Type bus is the most significant half).

Again, we'll point out that this operation can be in parallel to type checking on the Type board, memory access, and field isolation on the FIU board, as well as sequencing operations.

The symbolic encodings for this microinstruction are shown in Table 5–3.

The Val board uses the four cycle quarter signals, Q1~ through Q4~, as well as the cycle half signals, H1 and H2. These were discussed in Chapter 2.

The microinstruction is selected in the previous cycle. The microaddress comes from the microsequencer board and selects a location in the control store on

Table 5–2

Random Field Encodings, Val board

Encoding Name	Explanation
inc loop counter	increments the 10 bit loop counter
dec loop counter	decrements the 10 bit loop counter
condition to FIU	substitutes the selected Val condition for the least significant bit of the FIU bus. If all zeros are driven onto the other bits, this allows a boolean variable with the value of the condition to be used.
split C source	half of the data driven onto the C bus comes from the MUX and the other half comes from the FIU bus.
count zeros	activates the leading zero counter.
immediate op	causes the least significant byte of data on the B bus to come from the Val bus and all of the other bytes to come from the register file (as usual).
pass A high	the lower half of the ALU performs the operation selected by the microcode, the upper half of the ALU performs the PASS A operation.
pass B high	the lower half of the ALU performs the operation selected by the microcode, the upper half of the ALU performs the PASS B operation.
divide	activates several miscellaneous hardware support mechanisms to assist the microcode divide algorithm.
start multiply	causes the multiplier to latch the data currently being driven on the A and B buses for use as multiplier operands.
product left 16	forces the least significant bit of the multiplier output to be aligned on bit 47 instead of bit 63 (i.e., the output is shifted left 16 bits).
product left 32	forces the least significant bit of the multiplier output to be aligned on bit 31 instead of bit 63 (i.e., the output is shifted left 32 bits).

Table 5–3

Example Microinstruction, Val board

Field Name	Explanation
A Address	Mask register (can be any register in the file)
B Address	Val bus (the value coming from memory)
C Address	Result register (again any register in the file)
Register Frame	don't care
Mux Source	ALU unshifted
Random	No Operation
Multiplier A Source	don't care
Multiplier B Source	don't care
ALU Function	A AND B
C Source	MUX

each board. On the Val board, the outputs from the control store are latched in a microinstruction register (schematic pages 57 and 58) by UIR~.SCLK. This

starts the beginning of a microcycle.

In our example, we start just as the microinstruction register has been loaded with the microinstruction outlined above.

The Mask register from the register file is selected by the A Address lines (from the A Address field) during H1 (page 49) and latched into the A input latch at the end of Q2 (page 23). The A input latch drives the A bus during Q2 through Q4.

The B Address could select another register from the file if that was required. Instead, the B Address selects the memory data from the Val bus. The memory data is buffered by the Val bus transceiver (page 36). It drives the B bus during Q2 through Q4.

The ALU is active during the entire cycle (pages 32 and 33). Its inputs are valid during Q2 and its outputs (ALU bus and several conditions) become valid during Q3 or Q4, depending on the selected function.

The ALU bus is driven onto the C bus during all of H2 (pages 16 and 17) as selected by the C source field and the Mux source field.

The destination register is selected during all of H2 by the C Address field. Data from the C bus (and its logical equivalent, the C_BUF bus) is latched in the selected register at the end of Q4 by **A_RF.WE~** and **B_RF.WE~** if the microcycle has not been aborted.

This ends the example microinstruction. Now let's point out where the example is simplified.

There isn't one register file where two operands can be accessed at the same time. There are two register files (A and B) where each reads a separate operand and both are written with the same operand at the end of the cycle (pages 18 thru 21). The register files also maintain byte parity on their contents.

The register files contain 1024 locations which are broken down into "frames". There are 32 frames of 32 locations (0 thru 31). Frame 31 is reserved for the 16 general purpose registers which are implicitly addressed and for a "control stack accelerator" (CSA).

Fifteen locations of the register files are maintained as a cache of the top of the control stack. This is called the control stack accelerator (CSA). There isn't

much control logic on the Val board for the CSA, but there is the counters to maintain the top of stack location (page 52). Most of the control logic for the CSA is on the Type and FIU boards.

The multiply logic consists of two 64-bit input registers (A and B - pages 25 and 26), four 8 by 8 bit multiply chips arranged as a 16 by 16 multiplier (page 27), nibble propagate and carry propagate adders (page 28), and an output latch and driver (pages 29–31). The A and B input registers are loaded from the A and B bus respectively. The output latch drives the A bus. Steering logic (page 60) selects one of the four 16-bit quarters of each A and B registers for partial product generation. Additional steering logic directs output to maintain significance in the partial product. Full 64 bit by 64 bit multiplies take 16 cycles excluding any type checking or other setup.

The 10-bit loop counter can be loaded from the C bus and can drive the A bus (the least significant 10 bits of each - page 10). The loop counter can be used by microcode in any number of ways. The counter can be incremented by microcode and overflow and counter equals zero (page 34) can be used as a condition or as an address for the register file.

The leading zero counter (pages 11 and 12) can determine how many leading bits are zero of the ALU bus. Typically this is used for normalization of floating point numbers but can be used for anything. The count can drive the A bus.

There is a write data register that latches the contents of the Val bus during every memory write. This can be used as a C source to be loaded into a register in the register file at the end of any cycle. Byte parity from the Val bus is also retained.

The shift mux provides several possible sources for the C bus (pages 16 and 17). The mux, under the direction of the mux source microcode field, allows two shifted versions of the ALU output, the unshifted version of the ALU output, as well as the WDR to be driven onto the C bus.

The FIU bus may be driven or received on the Val board. The contents of the A bus may be driven onto the FIU bus and the contents of the FIU bus may be driven onto the C bus (page 38).

One of several conditions from the Val board may be selected for conditional sequencing. The microcode field to select the condition is latched on the microsequencer board. Several select lines come to the Val board via the backplane to

select three conditions (pages 34 and 35). These three conditions are sent back to the microsequencer via the backplane where final selection occurs.

Conditions can also be used for one other important operation: selecting one of two ALU operations. Particularly in divide operations, a condition is used to select addition or subtraction in the ALU.

One point about conditions: each has associated with it the notion of when in the microcycle the condition may become valid. This allows some conditions to affect the next microinstruction currently being read from control store (the early conditions), or conditions that can not be used until the following microinstruction (the late conditions). These are annotated on schematic page 35.

The ALU output may also be used as an address. It, under control of microcode, can be enabled onto the address bus (pages 40 and 41). Address parity is also maintained. Appropriate address constraints are imposed by this logic.

The remaining functional unit is the control store and microinstruction register (pages 53–60). There is a microaddress counter for loading the control store which is controlled by the diagnostic microprocessor. The control store is 40 16K by 1 static RAMs including a parity bit.

Diagnostic design

The diagnostic microprocessor on the Val board (the slave - pages 61–63) can read and write (directly or indirectly) all storage elements and test all functional units on the Val board. The slave can also provide status information whenever a parity error occurs on the Val bus, in the register files, in control store, etc.

The slave can read or write the following functional units:

- The 40-bit microinstruction register. This is accessible via 5 8-bit scan chains. The microinstruction register (UIR) is also the source of data when writing the control store. So, from this one register, (and with the diagnostic counter), the entire control store is accessible to the slave.
- A 14-bit diagnostic counter. This is a load or increment counter which can be used to address the control store. This register can not directly be read.

- The 72-bit write data register (WDR). This is accessible via 6 12-bit scan chains. It can be used as input and output for board experiments.

- The 4-bit CSA (control stack accelerator) offset register. This register points to the register in the register file which contains a copy of the top-of-stack. It is a write-only register and must be set during initialization or context-restore.

- An 8-bit parity error status chain.

- The 3 conditions that are sent to the sequencer board are also accessible.

The slave thus has the capability of loading control store with any microcode (diagnostic or otherwise), and loading the WDR with any data, and running the board a number of cycles and testing the WDR for results. Since the slave can provide any microcode, the slave can run experiments the exercise the board in exactly the same way as the real R1000 microcode.

Table 5–4 lists all of the stimuli that the slave can generate.

All of these stimuli are used in the experiments on the Val board. The set of experiments are described in a separate document:

`xx:<usd.value.diagnostic>experiments.text`

A separate set of experiments are collected into a program for diagnosing the entire board: the FRU diagnostic for the Val board. These experiments are described in a separate document:

`xx:<usd.value.fru>experiments.text.`

Table 5–4

DFSM stimuli, Val board

Signal	Function
cm.diag_on	Enables diagnostic control of the C mux and C source
diag_mux_sel	Selects the C mux for the C bus
bank_select	Most significant bit of DFSM state bit
val.diag_on	Enable the Val bus drivers while in diagnostic mode
fiu.diag_on	Enable the FIU bus drivers while in diagnostic mode
fsm_done	Signals the 8051 that the experiment is over
diag.wdr.s0 and s1	Selects the shift right, shift left (not used), load, or hold function of the WDR
a_l.diag_off	Disable A latch
force_pass_a	Forces the ALU function to be pass and forces the most significant 5 offset bits of the address bus to 1s
b_l.diag_off	Disable B latch
acs.diag_off	Disables all reads and writes to the A register file
bcs.diag_off	Disables all reads and writes to the B register file
diag.wdr.en	Enables diagnostic control of the WDR
diag_stop	Stops all writes to CSA, condition latches and zero counter
csa.diag.en	Enables clocking of CSA
novram.cs	Enables reading or writing the NOVRAM
diag.no_stop	Enables writing the NOVRAM and prevents outside events from stopping the Val board
cond.diag.en	Enables clocking of conditions
uir.diag_off	Forces UIR.SCLK off†
uir.diag_on	Forces UIR.SCLK on

† This is an active signal when the R1000 is in "run" mode. The UIR.SCLK signal is forced off during H1 and passes a 2X clock during H2. This effectively makes UIR.SCLK a Q4 signal.

Table 5-4 (continued)
DFSM stimuli, Val board

Signal	Function
b_o.diag_on	Disables microcode control of the B bus sources
c_d.diag_on	Disables the C bus to C_BUF bus driver
adr.diag_on	Disables microcode control of the address bus drivers
a_o.diag_on	Disables microcode control of the A bus sources
diag_mode	Disables microcode control of the Val and FIU bus drivers. Enables diagnostic control of the CSA control registers. Disables the latching the multiplier input latches.
freeze.en	Stops all microstate clocks and notifies other boards in the system that the Val board is frozen
pareg.sel0 and sel1	Selects the hold, right shift, load, or load if parity error occurred function of the parity error register
diag_write0 and 1	Controls writing the A and B register file while under diagnostic control
csa.diag0, 1, and 2	Controls the CSA control registers
scan_control	Reads the parity scan chain and the 3 condition bits into the 8051
wcs.diag.we	Enables writing the control store
scan_wdr	Reads the WDR scan chains into the 8051
read_diag_bus	Controls the direction of the diagnostic data bus; selects reading or writing
scan_uir	Reads the UIR scan chains into the 8051
nov_store.d	Writes the current contents of the RAM part of the NOVRAM into the permanent ROM part
trigger_scope	Provides a signal for triggering scopes or logic analyzers
loop.diag.ld	Loads the loop counter
loop.diag.ct	Increments the loop counter
loop.diag.en	Enables incrementing the loop counter
diag_cntr.en	Enables incrementing the diagnostic counter
diag_cntr.ld	Loads the diagnostic counter
uir.sel0	Selects the shift right or load function of the UIR
adr.diag_off	Enables the address bus drivers
diag_uadr.sel	Selects the diagnostic counter or the sequencer board's microaddress

Chapter 6: Type Board design

The Type board has one primary function: to perform all the type comparisons and operations within the instruction set. This includes things like class checks, privacy checks, of-kind checks, range checks, etc.

These calculations are done in parallel to all other functions on all the other boards in the system. In particular, the Val board operations are separately controlled and done in parallel with all Type board operations. The Val board can provide integer or floating point operations in parallel with the type operations being done on the Type board.

The Type board is completely controlled by microcode. It takes a microaddress from the microsequencer, looks up the contents of its control store at that address, and executes its portion of the microinstruction.

The following list is the set of functional blocks that make up the Type board. They are also shown on the block diagrams.

- 1K by 64 bit register file
- Specific hardware for type checking (of-kind, privacy, class)
- General purpose 64-bit ALU
- 64-bit Type half of write data register (WDR)
- Shift-mux for selecting write data for the register file
- 10-bit loop counter
- 47-bit control store with microinstruction register
- and the standard slave diagnostic microprocessor

The Type board also has four buses that connect the various blocks: ALU bus, A bus, B bus, and C bus.

Table 6–1

Microcode fields, Type board

Field Name	Bits	Explanation
A address	0–5	Specifies the A bus contents
B address	6–11	Specifies the B bus contents
Register frame	12–16	Selects one of 32 frames within the register file or the least significant 5 bits of a class literal
Class Literal	17–18	The most significant 2 bits of a class literal
Parity	19	The parity bit should be a 1 if all of the bits of the microword (on the Type board) are 1
Random	20–23	Selects one of 16 random operations including split bus operations, incrementing or decrementing the loop counter, and special class and privacy checks
C address	24–29	Selects the destination for the C bus
Privacy check	30–32	Selects what privacy check(s) is to be performed
Mux source	33	Selects the write data register or the ALU output for possible use on the C bus
ALU function	34–38	Selects one of 32 possible functions for the ALU
C source	39	Selects the Mux source or the FIU bus for the C bus
MAR control	40–43	Specifies the operation to be performed on the memory address register
CSA control	44–46	Specifies the operation to be performed on the control stack (both Type and Val halves)

That is the basic blocks you'll see on the first few pages of the schematics. How the Type board performs the type operations within the instruction set is really a function of microcode. We'll look at the microcode organization on the Type board briefly first. Then we'll go through the functional design of the Type board.

It should be pointed out that the Type and Val boards are very similar. The bus structure, register file organization, and many other functional blocks are the same on the two boards. The prime difference is the Type board has the type checking logic and the Val board has multiply logic.

Microcode organization

The Type board contains 47 bits of microcode including 1 parity bit. These are broken up into 13 fields. Table 6–1 describes each of these fields.

Table 6–2

Random Field Encodings, Type board

Encoding Name	Explanation
inc loop counter	increments the 10 bit loop counter.
dec loop counter	decrements the 10 bit loop counter.
split C source	half of the data driven onto the C bus comes from the MUX and the other half comes from the FIU bus.
check class A lit	check that the least significant 7 bits of the A bus are equal to the class literal. If not, generate a class check microevent.
check class B lit	check that the least significant 7 bits of the B bus are equal to the class literal. If not, generate a class check microevent.
check class A eq B	check that the least significant 7 bits of the A bus are equal to the least significant 7 bits of the B bus. If not, generate a class check microevent.
check class AB lit	check that the least significant 7 bits of the A bus and B bus are equal to the class literal. If not, generate a class check microevent.
pass A high	the lower half of the ALU performs the operation selected by the microcode, the upper half of the ALU performs the PASS A operation.
pass B high	the lower half of the ALU performs the operation selected by the microcode, the upper half of the ALU performs the PASS B operation.
carry in Q	selects the Q bit from the Val board to be the carry into the ALU (used when doing a divide).
write outer frame	loads the outer frame register with the data driven on the most significant half of the B bus.
set pass privacy	sets the Pass Privacy control bit in the privacy checking logic (forces the next cycle to pass a privacy check).
check class system	check that the least significant 7 bits of the B bus are equal to the class leteral. If not, generate a system class check microevent.
add dec 128	causes the INC A (INC B) ALU operation to increment (decrement) the value on the A bus by 128 rather than by 1.

The exact encodings are shown in the summary sheet attached to the front of the schematics. The specification for the Type board cotains a full description of all the encodings of all the fields.

The Random field is particularly distinct because its operations are not associated with each other (that's why it's called random). Table 6–2 provides a summary of all the encodings of the Random field.

One other area of interest that should be covered before getting into the hardware is what "events" can this board generate. Events, you will recall, are

Table 6–3

Microevents, Type board

Event Name	Explanation
Binary Equality	One or both of the operands does not allow assignment or equivalence operations
Binary Operation	One or both of the operands does not allow arbitrary binary operations
TOS Operation	The top-of-stack object does not allow any unary operations
TOS1 Operation	The object one below the top-of-stack does not allow any unary operations
Class	An object is not of the correct class for a particular operation
Check System	A system object (unaccessible to user programs) is not of the correct class for a particular operation

interrupts to the microsequencer that temporarily change the flow of microcode. The Type board only generates microevents and Table 6–3 shows them.

Functional Design

To show how the Type board operates, we'll make up a single, simple microinstruction and show how it routes data through the board. It won't show all of the paths nor all of the functional units. But it will give you an easy view of how the board operates in its simplest way.

First, a description of what this microinstruction is to do. We want to read an operand from memory, and a second operand already in a register, check some of the resulting bits for specific type information, and place the Type information from memory into another register. The value from memory comes via the Type bus which is the most significant half of the 128-bit data to or from memory (the Val bus is the least significant half).

Again, we'll point out that this operation can be in parallel to value operations on the Val board, memory access, and field isolation on the FIU board, as well as sequencing operations.

The symbolic encodings for this microinstruction are shown in Table 6–4.

Table 6–4

Example Microinstruction, Type board

Field Name	Explanation
A Address	Operand register (can be any register in the file)
B Address	Type bus (the value coming from memory)
C Address	Result register (again any register in the file)
Register Frame	don't care
Class Lit	don't care
Random	No Operation
Privacy Check	For Equality check
Mux Source	ALU
ALU Function	Pass B
C Source	MUX
MAR Control	No operation
CSA Control	No operation

The Type board uses the four cycle quarter signals, Q1~ through Q4~, as well as the cycle half signals, H1 and H2. These were discussed in Chapter 2.

The microinstruction is selected in the previous cycle. The microaddress comes from the microsequencer board and selects a location in the control store on each board. On the Type board, the outputs from the control store are latched in a microinstruction register (schematic pages 54–56) by UIR~.SCLK. This starts the beginning of a microcycle.

In our example, we start just as the microinstruction register has been loaded with the microinstruction outlined above.

The operand from the register file is selected by the A Address lines (from the A Address field) during H1 (page 45) and latched into the A input latch in Q2 (page 21). The A input latch drives the A bus during Q2 through Q4.

The B Address could select another register from the file if that was required. Instead, the B Address selects the memory data from the Type bus. The memory data is buffered by the Type bus transceiver (pages 32 and 33). It drives the B bus during Q2 through Q4.

The ALU is active during the entire cycle (pages 28 and 29). Its inputs are valid during Q2 and its outputs (ALU bus and several conditions) become valid during Q3 or Q4, depending on the selected function.

The ALU bus is driven onto the C bus during all of H2 (pages 16 and 17) as selected by the C source field and the Mux source field.

The destination register is selected during all of H2 by the C Address field. Data from the C bus (and its logical equivalent, the C_BUF bus) is latched in the selected register at the end of Q4 by A_RF.WE~ and B_RF.WE~ if the microcycle has not been aborted.

This ends the example microinstruction. Now let's point out where the example is simplified.

There isn't one register file where two operands can be accessed at the same time. There are two register files (A and B) where each reads a separate operand and both are written with the same operand at the end of the cycle (pages 18 thru 21). The register files also maintain byte parity on their contents.

The register files contain 1024 locations which are broken down into "frames". There are 32 frames of 32 locations (0 thru 31). Frame 31 is reserved for the 16 general purpose registers which are implicitly addressed and for a "control stack accelerator" (CSA).

Fifteen locations of the register files are maintained as a cache of the top of the control stack. This is called the control stack accelerator (CSA). The control logic for both the Val and Type board CSAs are on the Type board (pages 48 and 56). Additional logic for the CSA is on the FIU board.

The CSA control keeps track of which of the fifteen registers contains the top of the stack, which contains the bottom of the accelerated locations, and determines whether there are sufficient control stack locations locally to perform the next instruction (based on useage specified by the microsequencer board). Events are generated (on the microsequencer board) if there are insufficient CSA words to complete the instruction (CSA underflow) or too many CSA words locally that will require writing out to memory (CSA overflow).

The checker logic includes an outer frame register, privacy check logic, class check logic, of-kind check logic, and event generation logic (pages 23–27). Privacy checks make sure that an object is accessible meaning that the type of the object was not declared private. Class checks make sure that two objects are of the same type class such that operations between those two objects are allowed. Of-kind checks make sure that the two objects are of the same representation

class so that the operations make sense. These checks are required by the Ada rules on typing.

One or two types are presented on the A and B bus and this type checking logic determines if those two types are compatible for any given operation. The privacy, class_lit, and randoms microcode fields specify what type of operation(s) is planned and events are generated and sent to the microsequencer if the planned operation is not allowed. Certain conditions checked by this logic, may cause the processor to stop.

The 10-bit counter can be loaded from the C bus and can drive the A bus (the least significant 10 bits of each - page 10). The loop counter can be used by microcode in any number of ways. The counter can be incremented by microcode counter equals zero (page 30) can be used as a condition.

There is a write data register that latches the contents of the Type bus every cycle. This can be used as a C source to be loaded into a register in the register file at the end of any cycle. The Type bus and write data register maintain byte parity.

The shift mux provides one of two possible sources for the C bus (page 15). The mux, under the direction of the mux source microcode field, selects the ALU output or the WDR to be driven onto the C bus.

The FIU bus may be driven or received on the Type board. The contents of the A bus may be driven onto the FIU bus and the contents of the FIU bus may be driven onto the C bus (page 38).

One of several conditions on the Type board may be used on the microsequencer for conditionals. The microsequencer contains the microcode field that selects a condition in the processor. It sends out three select lines which is used on the Type board to select five of the 33 conditions on the board. The condition logic (pages 30 and 31) sends these five conditions to the microsequencer board where the final selection occurs.

Conditions can also be used for one other important operation: selecting one of two ALU operations. Particularly in divide operations, a condition is used to select addition or subtraction in the ALU.

The ALU output may also be used as an address. It, under control of microcode, can be enabled onto the address bus (pages 36 and 37). Address parity is also maintained. Appropriate address constraints are imposed by this logic.

The remaining functional unit is the control store and microinstruction register (pages 49–56). There is a microaddress counter (diagnostic counter) for loading the control store which is controlled by the diagnostic microprocessor. The control store is 47 16K by 1 static RAMs including a parity bit.

Diagnostic design

The diagnostic microprocessor on the Type board (the slave - pages 57–59) can read and write (directly or indirectly) all storage elements and test all functional units on the Type board. The slave can also provide status information whenever a parity error occurs on the Type bus, in the register files, in control store, etc.

The slave can read or write the following functional units:

- The 47-bit microinstruction register. This is accessible via 5 8-bit scan chains, and a 7-bit chain. The microinstruction register (UIR) can also be the source of data when writing the control store. So, from this one register, (and with the diagnostic counter), the entire control store is accessible to the slave.

- A 14-bit diagnostic counter. This is a load or increment counter which can be used to address the control store.

- The 72-bit write data register (WDR). This is accessible via 6 12-bit scan chains. It can be used as input and output for board experiments.

- An 8-bit CSA status scan chain. This is loaded and stored with the UIR chains.

- The 4-bit CSA (control stack accelerator) offset register. This register points to the register in the register file which contains a copy of the top-of-stack. It is a write-only register and must be set during initialization or context-restore.

- An 8-bit parity error status chain.

- The 6 microevents that can cause microcode interruptions.

Table 6–5

DFSM stimuli, Type board

Signal	Function
cm.diag_on	Enables diagnostic control of the C mux and C source
bank_select	Most significant bit of DFSM state bit
type.diag_on	Enable the Type bus drivers while in diagnostic mode
fiu.diag_on	Enable the FIU bus drivers while in diagnostic mode
fsm_done	Signals the 8051 that the experiment is over
diag.wdr.s0 and s1	Selects the shift right, shift left (not used), load, or hold function of the WDR
a_l.diag_off	Disable A latch
force_sp_hi	Forces the space address bits to 1s
b_l.diag_off	Disable B latch
tadr.diag_on	Disables microcode control of the address bus
acs.diag_off	Disables all reads and writes to the A register file
bcs.diag_off	Disables all reads and writes to the B register file
diag.wdr.en	Enables diagnostic control of the WDR
diag_stop	Stops all writes to CSA, condition latches and zero counter
csa.diag.en	Enables clocking of CSA
novram.cs	Enables reading or writing the NOVRAM
diag.no_stop	Enables writing the NOVRAM and prevents outside events from stopping the Type board
cond.diag.en	Enables clocking of conditions
uir.diag_off	Forces UIR.SCLK off†
uir.diag_on	Forces UIR.SCLK on

† This is an active signal when the R1000 is in "run" mode. The UIR.SCLK signal is forced off during H1 and passes a 2X clock during H2. This effectively makes UIR.SCLK a Q4 signal.

- The 5 conditions that are sent to the microsequencer board are also accessible.

The slave thus has the capability of loading control store with any microcode (diagnostic or otherwise), and loading the WDR with any data, and running the board a number of cycles and testing both conditions and the WDR for results. Since the slave can provide any microcode, the slave can run experiments the exercise the board in exactly the same way as the real R1000 microcode.

Table 6–5 lists all of the stimuli that the slave can generate.

All of these stimuli are used in the experiments on the Type board. The set of experiments are described in a separate document:

Table 6–5 (continued)

DFSM stimuli, Type board

Signal	Function
b_o.diag_on	Disables microcode control of the B bus sources
c_d.diag_on	Disables the C bus to C_BUF bus driver
a_o.diag_on	Disables microcode control of the A bus sources
diag_mode	Disables microcode control of the Type and FIU bus drivers. Enables diagnostic control of the CSA control registers.
freeze.en	Stops all microstate clocks and notifies other boards in the system that the Val board is frozen
pareg.sel0 and sel1	Selects the hold, right shift, load, or load if parity error occurred function of the parity error register
diag_write0 and 1	Controls writing the A and B register file while under diagnostic control
csa.diag0, 1, and 2	Controls the CSA control registers
scan_control	Reads the parity scan chain, the 5 condition bits, and the 6 microevents into the 8051.
wcs.diag.we	Enables writing the control store
scan_wdr	Reads the WDR scan chains into the 8051
read_diag_bus	Controls the direction of the diagnostic data bus; selects reading or writing
scan_uir	Reads the UIR scan chains into the 8051
nov_store.d	Writes the current contents of the RAM part of the NOVRAM into the permanent ROM part
trigger_scope	Provides a signal for triggering scopes or logic analyzers
loop.diag.ld	Loads the loop counter
loop.diag.ct	Increments the loop counter
loop.diag.en	Enables incrementing the loop counter
diag_cntr.en	Enables incrementing the diagnostic counter
diag_cntr.ld	Loads the diagnostic counter
uir.sel0	Selects the shift right or load function of the UIR
adr.diag_off	Enables the address bus drivers
diag_uadr.sel	Selects the diagnostic counter or the sequencer board's microaddress

xx:<usd.type.diagnostic>experiments.text.

A separate set of experiments are collected into a program for diagnosing the entire board: the FRU diagnostic for the Type board. These experiments are described in a separate document:

xx:<usd.type.fru>experiments.text.

Chapter 7: Field Isolation Board design

The Field Isolation Unit (FIU) board has several functions:

- provide 1- to 64-bit insertion and extraction into arbitrary up-to-128-bit data
- provide memory control and monitor functions
- provide control stack accelerator control and monitor functions

This really is the set of things that the FIU board does. But that isn't necessarily "how" the FIU board does them. The following list is the set of functional blocks that make up the FIU board. They are also shown on the block diagrams.

- 128-bit input, 64-bit output rotator (barrel shifter)
- 64-bit merge data register (MDR)
- 128-bit merger
- 64-bit Val bus assembly register (VAR)
- 64-bit Type bus assembly register (TAR)
- Val, Type, and FIU bus transceivers
- Parameter and field control
- Memory address register (MAR)
- Refresh counter
- Memory state and control
- CSA monitor state
- 39-bit control store and microinstruction register

- and the standard slave diagnostic microprocessor

The FIU board also has seven buses that connect the various blocks: FI bus, VI bus, VO bus, TI bus, TO bus, RDATA bus, and MAR bus. The MAR bus is broken down into several sections: the 3 MAR.SPACE bits, the 32 MAR.NAME bits, and the 32 MAR.OFFSET bits.

Those are the basic blocks you'll see on the first few pages of the schematics. How the FIU board performs those functions within the instruction set is really a function of microcode. We'll look at the microcode organization on the FIU board briefly first. Then we'll go through the functional design of the FIU board.

Microcode organization

The portion of the microinstruction contained on the FIU board is 39 bits wide, including a parity bit. This is broken into 17 fields. Table 7–1 provides the breakdown of these bits and a brief description of each field. The specification for the FIU board provides a more complete description.

One field, in particular, provides insight into what the board is capable of performing. This field controls the operation of the rotator and merger as shown in Table 7–2.

The FIU board generates four microevents as shown in Table 7–3.

In fact, the last three microevents are grouped under a common event called "memory exception". The microcode responding to the memory exception must interogate status bits to determine which event actually occurred.

The FIU board also generates one macroevent, Refresh Memory. Memory is refreshed in a single burst every 2 milliseconds by microcode. A counter on the board requests the microcode to refresh all of memory whenever the counter overflows.

Table 7–1

Microcode fields, FIU board

Field Name	Bits	Explanation
Offset literal	0–6	Specifies a bit offset within the 128-bit input of the rotator of the beginning of the field to insert or extract.
Length/fill literal	7–13	Specifies a bit length and fill mode of the field to insert or extract.
Length/fill register control	14–15	Specifies to load and from where to load the length/fill register.
Operation select	16–17	Selects one of four operations; see Table 7-2.
Merge vmux select	18–19	Selects one of three sources for one input to the merger.
Fill mode source	20	Selects either the fill register or the fill literal to specify the fill mode.
Offset register source	21	Selects either the address bus or the offset literal to load into the offset register.
TI VI source	22–25	Selects the source(s) for the TI and VI buses.
Load offset register	26	Selects load or hold for the offset register.
Load VAR	27	Selects load or hold for the Val bus assembly register.
Load TAR	28	Selects load or hold for the Type bus assembly register.
Load MDR	29	Selects load or hold for the Merge data register.
Memory start	30–34	Selects an operation for memory.
Rdata bus source	35	Selects the MDR or the rotator output for the Rdata bus.
Parity	36	This bit should be a 1 if all of the other bits in the microword on the FIU board are 1.
Length Source	37	Selects either the length literal or the length register to specify the length.
Offset source	38	Selects either the offset literal or the offset register to specify the offset.

Functional Design

Since there are three functional areas on the FIU board (Rotator/Merger, Memory Monitor, and CSA Monitor), we will divide up the discussion along those line.

Rotator/Merger

As with other boards, we'll make up a simple, single microinstruction and show how it routes data through the Rotator/Merger. It won't show all of the data paths nor all of the functional units. But it will give you an easy view of how the Rotator/Merger works.

Table 7–2

Operation Select Encodings, FIU board

Encoding	Explanation
Extract	Specifies that a bit string of some length (0–64 bits) and at some offset within the 128-bit input word, right-justified in a 64-bit halfword, filled with zeros or sign-extended, and (presumably) stored in some register on another board or into the assembly registers on the FIU board.
Insert	Specifies that a bit string of some length (0–64 bits) is to be inserted into a 128-bit word at some offset.
Insert first	Specifies that some most significant portion of a bit string of some length (0–64 bits) is to be inserted into the least significant portion of the 128-bit word at some offset. Assumes that the remaining least significant portion of the bit string will be inserted into the next word by the next operation: Insert last.
Insert last	Specifies that some least significant portion of a bit string of some length (0–64 bits) is to be inserted into the most significant portion of the 128-bit word. Assumes that the remaining most significant portion of the bit string has already been inserted into a previous word by the previous operation: Insert first.

Table 7–3

Microevents, FIU board

Event Name	Explanation
Page crossing	The MAR has been incremented across a half page boundary. Note the half page boundary. Note also no memory start is required to get this exception.
Cache miss	One of three conditions: A memory start was requested for a location that is not in main memory; a memory write to a read-only memory page; or a memory start to a page that is in the loading state.
Out of range	A control stack address was requested beyond valid control stack addresses. In other words, a memory start to a control stack location above the current top-of-stack.
Scavenger trap	No longer used.

First, a description of what this sample microinstruction is to do. A record in memory contains a 30-bit field beginning at bit 54. The 128-bit word that contains that field is being read from memory and will be placed in a register on the Val board. This sample microinstruction will take the 128-bit word from memory, remove the requested 30 bits, right-justify and sign-extend it, and send it via the FIU bus to the Val board.

The symbolic encodings for this microinstruction are shown in Table 7–4. The fields have been rearranged for clarity of purpose.

As with all other boards, the microinstruction is selected and read from control store in the previous cycle (schematic pages 35–38). The outputs of the control

Table 7–4

Example Microinstruction, FIU board

Field Name	Explanation
Operation select	Extract
Offset literal	54 – position of first bit to extract
Offset source	Offset literal
Offset register source	Don't care
Load offset register	Don't care
Length/fill literal	29 – positive means sign fill; length-1
Fill mode source	Fill literal
Length source	Length literal
Length/fill register control	Don't care
TI VI source	Type/Val – use Type and Val buses as input to rotator
Merge vmux select	Sign bit – source of sign extension bits
Rdata bus source	Rotator – use rotator output as input to Merger
Load VAR	Don't care
Load TAR	Don't care
Load MDR	Don't care
Memory start	Don't care
Parity	as appropriate

store are latched in a microinstruction register (page 39) by UIR.CLK. This starts the beginning of a microcycle.

The Type bus and Val bus tranceiver (pages 6–9) receive that memory data from the Type and Val buses near the beginning of the cycle. (The memory data is placed on these buses by microcode on another board.) This data is enabled onto the TI bus and VI bus for the duration of the microcycle.

The 128-bit input of the rotator is the TI and VI buses. This input is shifted, in three tiers, to the right 44 bits (pages 17–20). The rotate amount (in the example 44) is calculated (page 33) from the length and offset specified in the microcode (pages 31 and 32). The sign bit of the 30-bit result is also extracted (page 21). The 30-bit result appears in the least-significant bits of the 64-bit rotator output (the RDATA bus).

The merger, in this case, is used to sign-extend the output of the rotator. The merger takes the output of the rotator (the RDATA bus) and the sign bit replicated in all bit positions from the Merge Vmux as data inputs. Control logic

generates the start bit position and end bit position of valid data (page 34). The start bit and end bit positions are used by the merger to generate a merge mask. The merge mask selects which of the two inputs is selected in each bit position (page 25 and 26).

In the example, the start bit is 98 and the end bit is 127. The merge mask in the example is zero in positions 0 through 97 and is one in positions 98 through 127. This selects the TI bus in positions 0 through 63 (which doesn't matter since the upper 64 bits of the merger aren't used), the sign bit in positions 64 through 97, and the rotator output in positions 98 through 127.

The least-significant 64 bit output of the merger is enabled onto the FIU bus to complete the example (page 10 and 11). From there, another board will enable it into a register.

This completes the example. It and several other example operations are shown in Figure 7–1.

Now let's go back over the functional blocks describing all of the capabilities.

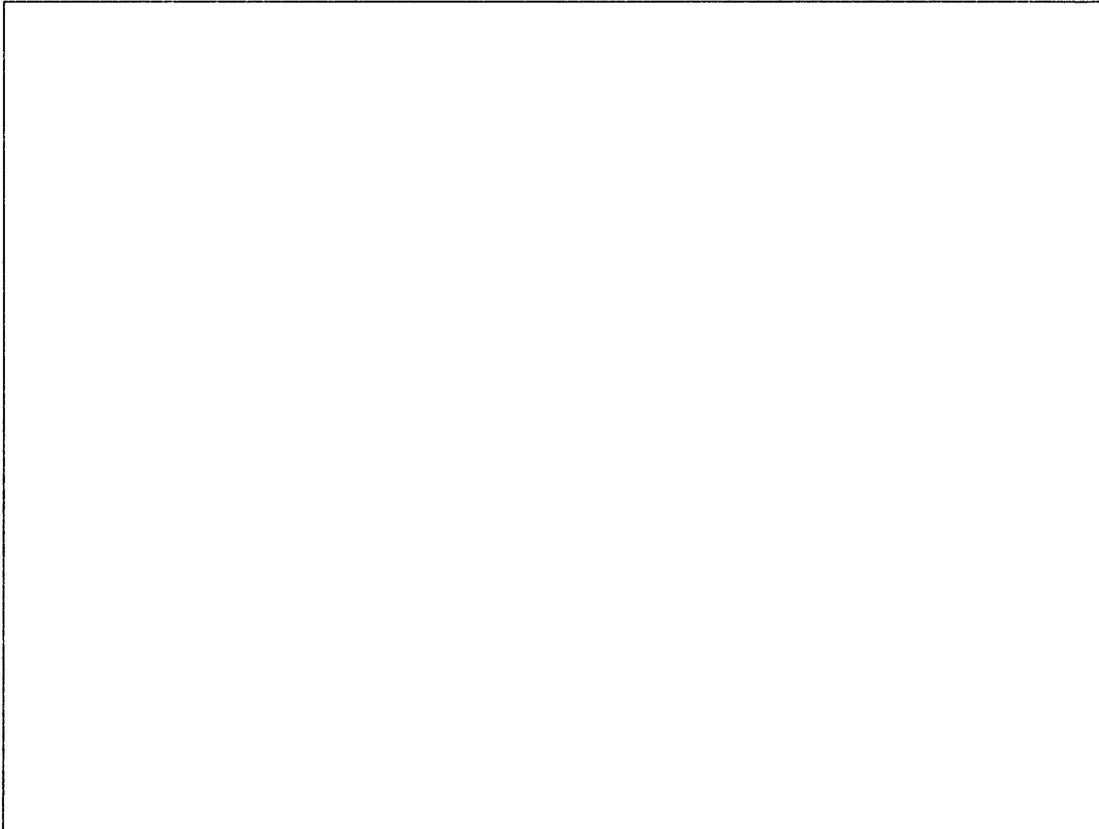
The Type and Val bus transceiver maintains and monitors byte parity on each bus. The transceiver connects these buses to the TI and VI buses. These internal buses have sources of the Type and Val bus transceiver as well as the FIU TV driver (page 12) which brings data in from the FIU bus, the MAR, the frame (or physical memory) address, and the Type and Val assembly registers (TAR and VAR – pages 13 and 14). The internal buses have destinations of the rotator, the merge vmux, and the merger.

The rotator and merger both take control information from the field control logic (pages 33 and 34). This logic generates the rotate amount for the rotator and the start and end bit positions for the merge mask. These three values are generated based on the operation selected as shown in Table 7–5.

The field control logic gets offset and length information from the parameter generation logic (pages 31 and 32). This logic generates the length, offset and fill parameters. A length/fill register can be loaded from the length literal field in microcode, from the TI bus, or from the VI bus. An offset register can be loaded from the offset literal field in microcode, or from the Address bus. Either or both registers can be used in any later cycle instead of the respective literal fields from that current microinstruction.

Figure 7–1

Examples of Rotator/Merger Operation



The 64-bit Merge Data Register (MDR – page 22) is used to hold data from the rotator for merging into data in another cycle. It can be loaded in any cycle with data from the rotator and used in any later cycle as an input to the merger.

The Merge Vmux (pages 23 and 24) selects one 64-bit source for the merger. Its inputs are the FI bus (an internal version of the FIU bus), the VI bus, or the sign bit of the rotator output which has been replicated in all 64 bit positions. The selection is made specifically by microcode.

The merger (pages 25–30) has two 64-bit halves: the Val half, and the Type half. Each half is actually 64 2-to-1 multiplexors. Each multiplexor is controlled by a unique bit in the 128-bit merge mask. The merge mask is generated from the start bit and end bit positions supplied by the field control logic.

The Val half selects bits from either the Merge Vmux output or the RDATA bus.

Table 7–5

Field control generation

Operation	Value	Calculation
Insert	Rotate Amount Start bit End bit	(length + offset)mod 64 (offset)mod 128 (offset + length - 1)mod 128
Insert First	Rotate Amount Start bit End bit	not used offset 127
Insert Last	Rotate Amount Start bit End bit	not used 0 (offset + length - 1)mod 128
Extract	Rotate Amount Start bit End bit	(-(offset + length))mod 64 128 - length 127

Its output can be loaded into the Val assembly register or driven onto the FIU bus. The Val assembly register, in turn, supplies the VI bus.

The Type half selects bits from either the TI bus or the RDATA bus. Its output can be loaded into the Type assembly register. The Type assembly register, in turn, supplies the TI bus.

Memory Monitor

The memory monitor is the control center for the entire set of memory boards. It:

- maintains the memory address register (MAR),
- controls the operation of the memory boards based on microcode on the FIU board and events throughout the system,
- refreshes the entire memory set,
- and watches for pages of memory that are not in the memory but are instead on disk.

The memory monitor has three primary sources of information to perform these functions; the memory address register (MAR),, the memory status signals, and

Table 7–6

Memory Address Register

Field Name	Bits	Explanation
Refresh Interval	0–15	Specifies the number of cycles which must elapse before a Refresh Memory macro-event is requested – page 60
Refresh Window	16–31	Specifies the number of cycles which must elapse between the requesting of a Refresh Memory macro-event and the error condition Refresh Machine Check which freezes the processor – page 60
Memory State	32–35, 37–40	Memory state bits: Scavenger trap (no longer used); CSA out-of-range; page crossing; cache miss; physical last; write last; MAR modified; incomplete memory cycle – page 44
Fill mode register	36	The fill mode as explained in the Rotator/Merger section selects the type of fill: zero or sign-extend page 32
unused	41–42	
Length Register	43–48	The length register as explained in the Rotator/Merger section contains the length-1 of the field requested – page 32
unused	49–60	
Space	61–63	Specifies one of seven memory spaces – page 53
Name	64–95	Specifies a segment name – page 53; sometimes broken into the following two fields
Name – Segment	64–87	
Name – VPID	88–95	Selects a virtual processor
Offset	96–127	Specifies a bit offset within the named segment – page 53; usually broken into the following fields:
Offset – Page	96–114	Selects a page within the named segment
Offset – Word	115–120	Selects a word within the page
Offset – Bit	121–127	Selects a (initial) bit within the word

the memory start field of microcode. The (MAR) is shown in Table 7–6 with the page number in the schematics the field appears on. The memory status lines include the four memory board hit lines which indicate that which, if any, memory board has the requested location, and the page state bits of the requested page which indicate what state the page is in. The memory start field includes microorders for starting logical or physical memory reads or writes, checking, reading or writing tags, conditional starts, and page-mode starts.

The MAR is controlled by a field of microcode which is contained in the control store on the Type board (see the chapter on the Type board). Principle functions include loading or incrementing the MAR. Incrementing the MAR is used for page mode reads and writes (pages 44 and 54).

The memory monitor communicates with the memory boards principly via four control signals, **F.MEM_CTL0** through **F.MEM_CTL3** (page 42). These four signals

select one of the 16 functions that the memory boards can perform (see the chapter on the Memory board).

The memory boards communicate with the memory monitor principly via four hit signals, F.BOARD_HIT0 through F.BOARD_HIT3 (page 43). These signals enable the memory monitor to know whether the requested memory location is in memory or not. If it is not, the memory monitor signals a memory exception microevent, F.MEM_EXCEPTION~ (page 43 – also called a "cache miss").

The memory monitor takes the 5-bit memory start field from the microinstruction register, and inputs it into a state machine (page 42). The state machine also uses the two abort signals, LATE_ABORT and EARLY_ABORT. These abort signals are asserted by the microsequencer board whenever a memory operation may have been started but should be aborted (see the chapter on the Microsequencer board). Both are asserted by the FIU board whenever a microevent occurs.

The state machine generates two state bits, the four control signals that go to memory, a continue signal which effects a page mode read or write, and several other signals.

The page mode operations are initiated with the signal F.CONTINUE~ (page 42). This tells the memory boards that the same operation should be done again at the new (presumably incremented) memory address. Page crossings must be detected by the memory monitor to prevent memory addresses from incrementing across page boundaries as that might result in a page fault. Page crossings (actually done on half page boundaries) cause a microevent (page 43).

The memory monitor can also generate a frame address (or physical address). The hash function that the memory boards use to translate logical memory addresses into physical addresses is duplicated via a lookup table (page 55). The lookup table is a pair of RAMs that are loaded by the diagnostic processor.

The memory monitor also has a scavenger RAM (page 61). This was intended to provide a garbage collection scheme for the memory manager. However, that scheme was abandoned and the scavenger RAM is no longer used.

CSA Monitor

The Control Stack Accelerator (CSA) is a set of 15 registers on both the Type and Val boards that maintain a set of the top few locations on the control stack. The CSA monitor must watch for memory accesses that are contained in the CSA and direct them to the registers on the Type and Val boards instead of to memory. It also must perform several operations in adjusting the CSA as in removing several entries or adjusting for CSA overflow or CSA underflow.

The simplest case to study is the watching for memory addresses that are in the CSA. It is important that references to memory locations that are actually in the CSA be caught since the CSA will contain more up-to-date data than the memory locations.

The CSA monitor maintains a CTOP register (actually a counter – page 46) which contains the address of the top-of-stack. The memory address (pages 53 and 54 – see discussion under memory monitor above) is subtracted from the CTOP to produce a difference (pages 49 and 50). This difference is compared to the number of valid entries (NVE) in the CSA, which is maintained by the Type board (page 50). The comparison yields CSA_HIT which is an input to the memory monitor.

CSA_HIT has the effect, via logic on several boards, of forcing data onto the Type and Val buses from a selected CSA location in the Type and Val board's register files. This data is used instead of the data from memory.

The CTOP register must be decremented and incremented as necessary to keep track of the top-of-stack address. This is done by incrementing or decrementing the register as directed by the Type board.

There is one other operation, called a "pop down to", that the CSA monitor must participate in. The pop-down-to operation is used to remove a number of locations from the top-of-stack, typically removing temporary variables from the stack when returning from a subroutine. Microcode specifies how many locations to remove in a two step process, but microcode does not know whether the new top-of-stack is already in the CSA. The CSA monitor, using the same facilities as discussed above, can determine this.

An additional register is used: the OLD_CTOP register (also called the Pop Down register). This register is loaded with the old CTOP value in the first step of the pop-down-to process (page 46). Then the old CTOP value is used instead of the MAR during the subtraction in the second cycle of the process (page 48). The difference between that and the number of valid entries, if positive, is the new

number of valid entries after the pop-down-to. This value is sent to the Type board to update the number of valid entries along with CSA_HIT which indicates that the CSA contains the new top-of-stack.

Diagnostic design

The diagnostic processor (slave) on the FIU board can read and write, directly or indirectly, most storage elements on the board. This is used to verify the correct operation of the board as well as provide status when errors occur. The slave is on pages 64–66.

The slave can read or write the following set of functional units:

- The 64-bit Merge data register. This is accessible via 8 serial scan chains.
- The 4-bit MAR control field. This field is normally supplied by the Type board but can be supplied by the slave.
- The 3-bit CSA control field. This field is normally supplied by the Type board but can be supplied by the slave.
- The 39-bit microinstruction register. This is accessible via 3 8-bit chains, a 7-bit chain, a 6-bit chain, and a 2-bit chain.

  The microinstruction register can also be used to load a control store location. This, along with the control addressing via the refresh counter, allows the slave to read and write the control store.
- The Space, Name, and most-significant 25 Offset bits. These 60 bits (all but the least-significant 7 bit-offset bits) of the MAR are accessible via 7 8-bit chains and a 4-bit chain.
- The Scavenger RAM data. An addressed byte from the scavenger RAM can be read or written. (The scavenger RAM is addressed by the most-significant 9 bits of the MAR.
- A read-only bit from the MAR control PROM.

- An 8-bit chain which includes parity error bits from the Type, Val, FIU, and Address buses, plus the refresh machine check, the microaddress parity error, the microinstruction parity error, and the scavenger RAM parity error.
- The two read-only conditions from the the FIU board.

The NOVRAM is addressed by 8 bits from the Merge data register. It is read or written with data directly from the slave.

The slave can specify any microinstruction and can specify microcode which normally originates on another board. It also can specify many diagnostic functions and controls not available via microcode. These capabilities allow the slave to exercise the board at full speed in the same way microcode does. These capabilities are shown in Table 7–7.

Table 7–7
DFSM stimuli, FIU board

Signal	Function
Diag_WCS.WE	Write the addressed control store location
Diag_Hash.CS	Enable the Hash function RAMs
Diag_Hash.WE	Write the addressed hash function RAM location
Diag_CLKSTP.EN	Enable diagnostic stopping of the state clocks
Diag_CLKSTP	Stop the state clocks
Diag_CNT.EN	Enable diagnostic incrementing of the refresh counters
Diag_CNT.DIS	Disable all incrementing of the refresh counters
Diag_Load_CNT	Load refresh counters from TI bus
Diag_MEMABORT	Abort all memory operations in progress
Diag_MEMHOLD	Hold all memory state
Diag_MCTL.EN	Enable sending all memory control signals
Diag_ADR.SEL	Selects the normal microaddress or the refresh counter for addressing control store
MAR.SEL0 and 1	Selects load, shift left, shift right, or hold function of the MAR
Scan_MAR	Enables MAR scan chains onto the diagnostic data bus
Event_CLKSTP.EN	Enable memory events from stopping processor
Diag.Fpar	Force a parity error in the FIU bus parity detectors
Diag.Apar	Force a parity error in the Address bus parity detectors
MDR.SEL0 and 1	Selects load, shift left, shift right, or hold function of the Merge data register
Scan_MDR	Enables MDR scan chains onto the diagnostic data bus
UIR.mode	Enables load or shift right functions of the microinstruction register
Scan_UIR	Enables UIR scan chains onto the diagnostic data bus

Table 7-7 (continued)
DFSM stimuli, FIU board

Signal	Function
Dfreeze.FSM	Freeze the entire processor
Dsync.FSM	Sync the entire processor
Sel_Diag_cntl	Enable diagnostic control of the Val, Type, FIU, and Address bus drivers
Diag_TV.EN	Enable Diagnostic control of the TI and VI bus sources
Diag_T.EN	Drive the Type bus
Diag_V.EN	Drive the Val bus
Diag_F.EN	Drive the FIU bus
Diag_ADR.EN	Drive the address bus
FSM_done	Signals the slave that the DFSM command is complete
Sync_Frez.EN	Enable control of sync or freeze
Scan_misc	Enables the parity scan chain, the bit from memory monitor control, the FIU board code, and the two FIU conditions onto the diagnostic data bus
Read_Diag_bus	Enables data into the slave
NOVRAM.WE	Write the addressed location in the NOVRAM
NOVRAM.CS	Enable reading or writing the NOVRAM
NOVRAM.STORE	Store all locations in the NOVRAM in the ROM portion
DT.EN	Trigger a state change in the DFSM
PAREG.SEL0 and 1	Select load, shift left, shift right, or hold function in the parity error register
Scope_Sync	Backpanel pin for triggering scopes or logic analyzers
UIRCLK.EN	Enable clocking new microinstructions into the microinstruction register
My_freeze.EN	Freeze the FIU board only
Q4_disable	Disable all Q4 state clocks
Set_State0	Force the DFSM command into the upper set of 8 states

Chapter 8: Sysbus Board design

The sysbus board has several functions:

- List'em

This really is the set of things that the sysbus board does. But that isn't necessarily "how" the sysbus board does them. The following list is the set of functional blocks that make up the sysbus board. They are also shown on the block diagrams.

- List'em

That is the basic blocks you'll see on the first few pages of the schematics. But that isn't the "how" either. Unfortunately, we have to use words to describe how the board does the first list using the second list.

Functional Design

- follow simple operations of board first

- diverge into more complex operations

- refer to schematics occasionally (with page and signal name)

Microcode design

- organization of microcode fields on this board
- macro and micro events generated
- refer to spec, microsim, or microcode if appropriate

Diagnostic design

- design of microprocessor on this board (how different from others)
- set of experiments on this board
- cross reference to listings occasionally

Chapter 9: I/O Adapter Board design

The I/O Adapter (IOA) board has several functions:

- List'em

This really is the set of things that the IOA board does. But that isn't necessarily "how" the IOA board does them. The following list is the set of functional blocks that make up the IOA board. They are also shown on the block diagrams.

- List'em

That is the basic blocks you'll see on the first few pages of the schematics. But that isn't the "how" either. Unfortunately, we have to use words to describe how the board does the first list using the second list.

Functional Design

- follow simple operations of board first

- diverge into more complex operations

- refer to schematics occasionally (with page and signal name)

Microcode design

- organization of microcode fields on this board
- macro and micro events generated
- refer to spec, microsim, or microcode if appropriate

Diagnostic design

- design of microprocessor on this board (how different from others)
- set of experiments on this board
- cross reference to listings occasionally

Chapter 10: Examples of Operations

- run through power-up sequence
- show example of instruction execution (from DRS's patent discussion)

Draft Status

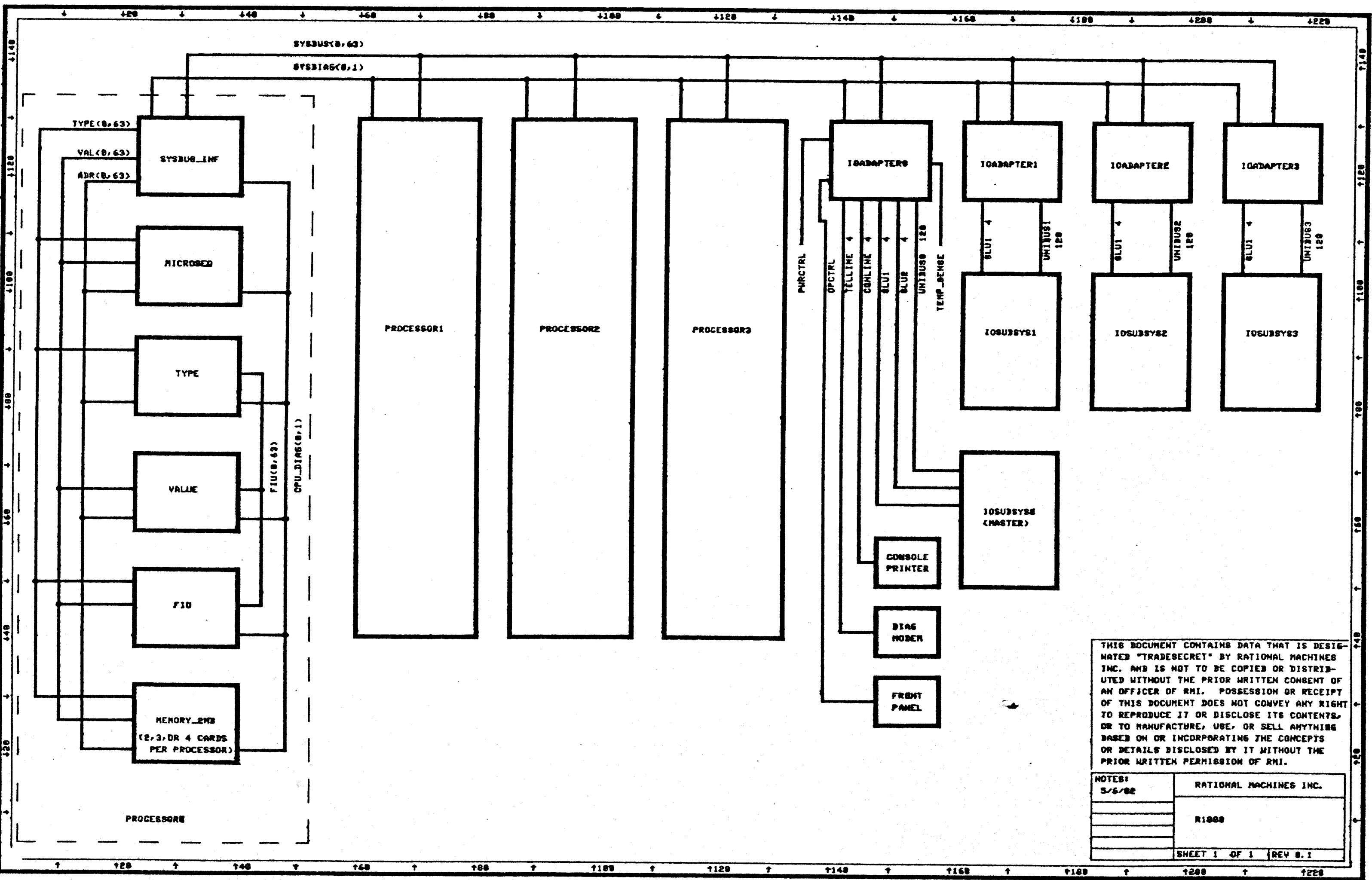
This draft outlines the entire book but only fills out only some of the chapters. The book is being developed even as you read this, though there are higher priority things I need to work on. As you read this book, think about whether it covers all that you think a new person here should know to a reasonable level of detail.

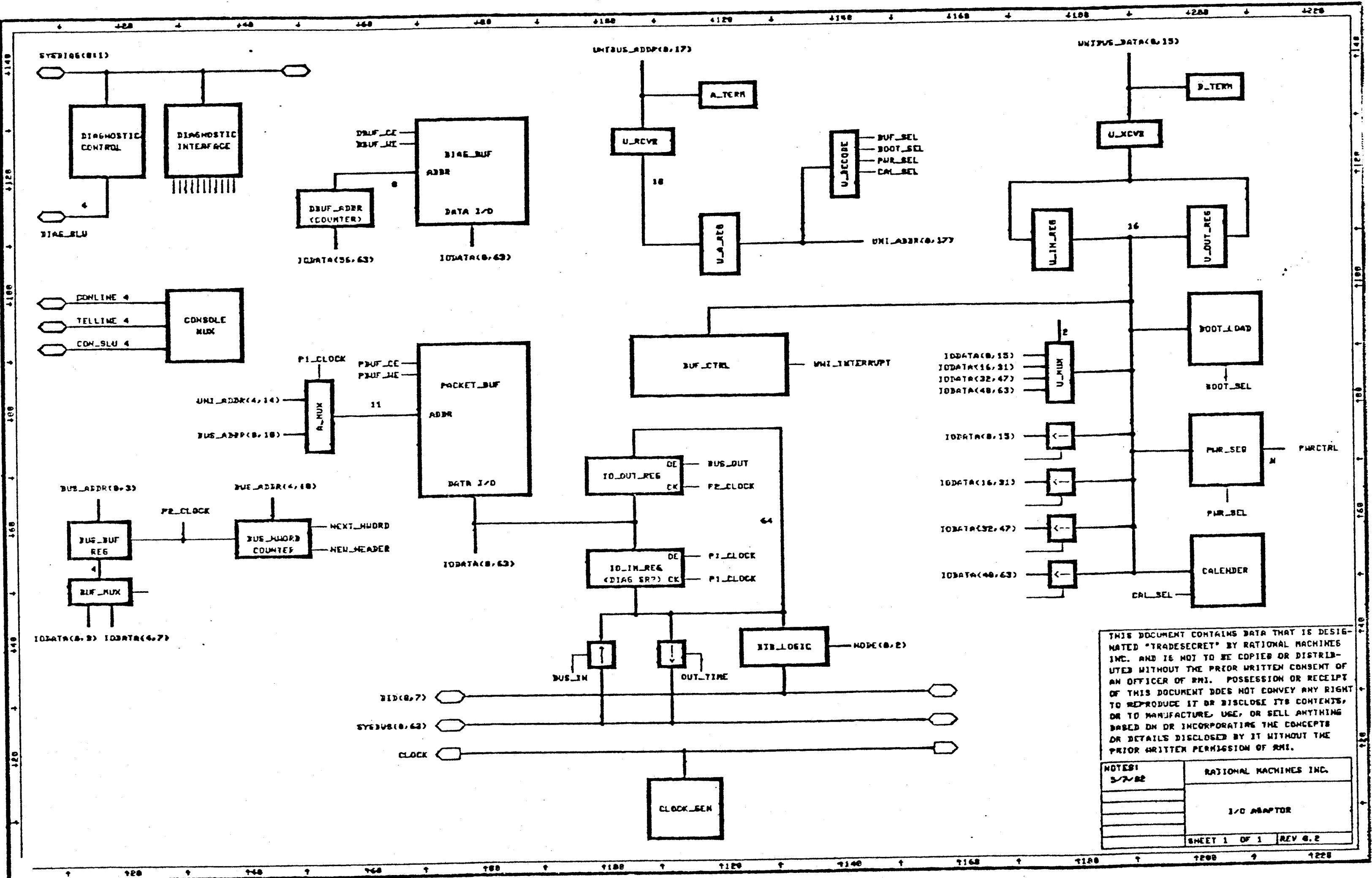
This draft is being used to test TEX macros as well. Please excuse TEX's (my) mistakes.

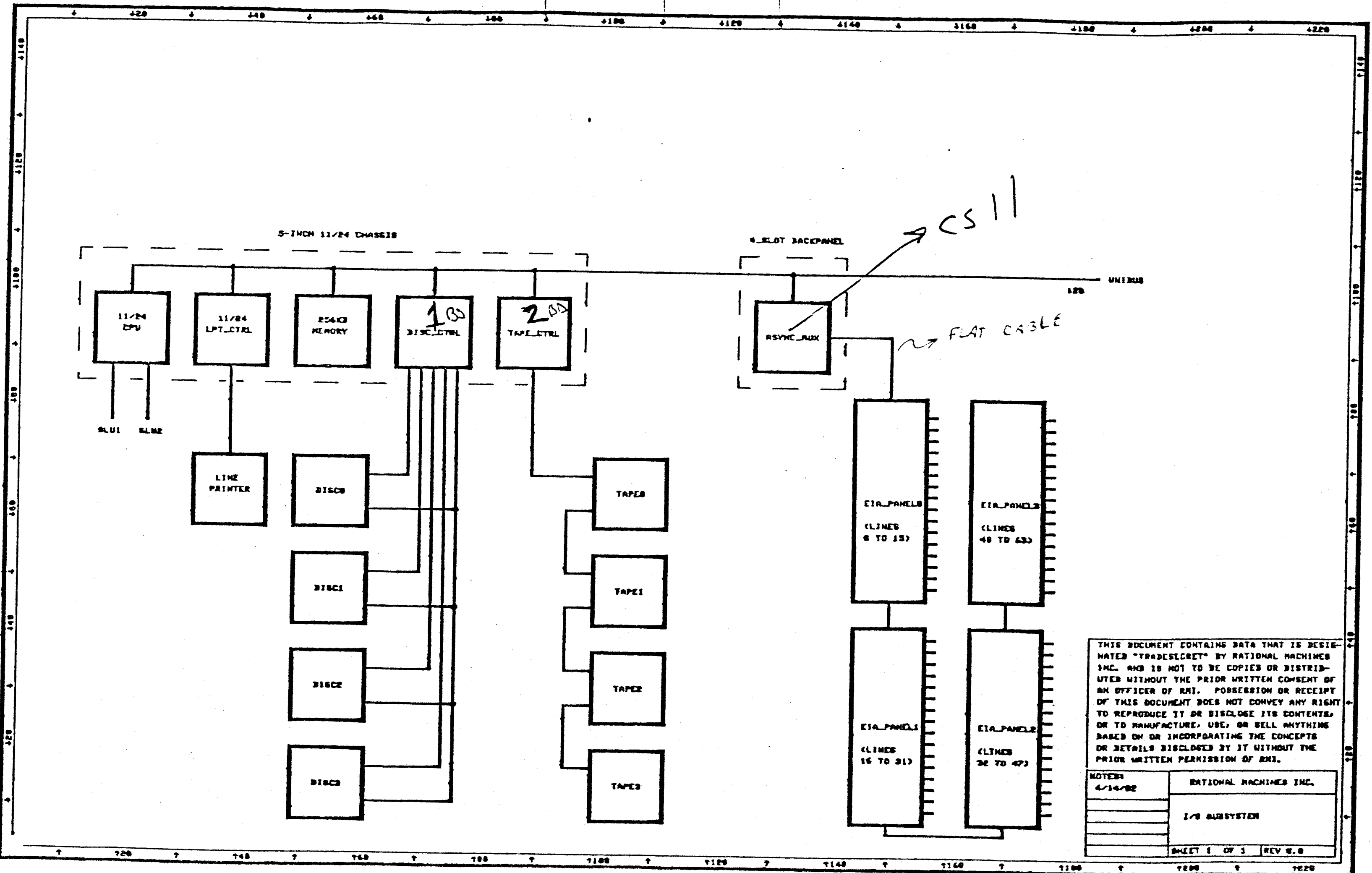
Please notice the words in the upper right hand corner of most pages like the one on this page. They are the words that appear in the index. If there is a word on any page that has been overlooked (or one that should have been) please mark such corrections on the list in the corner.

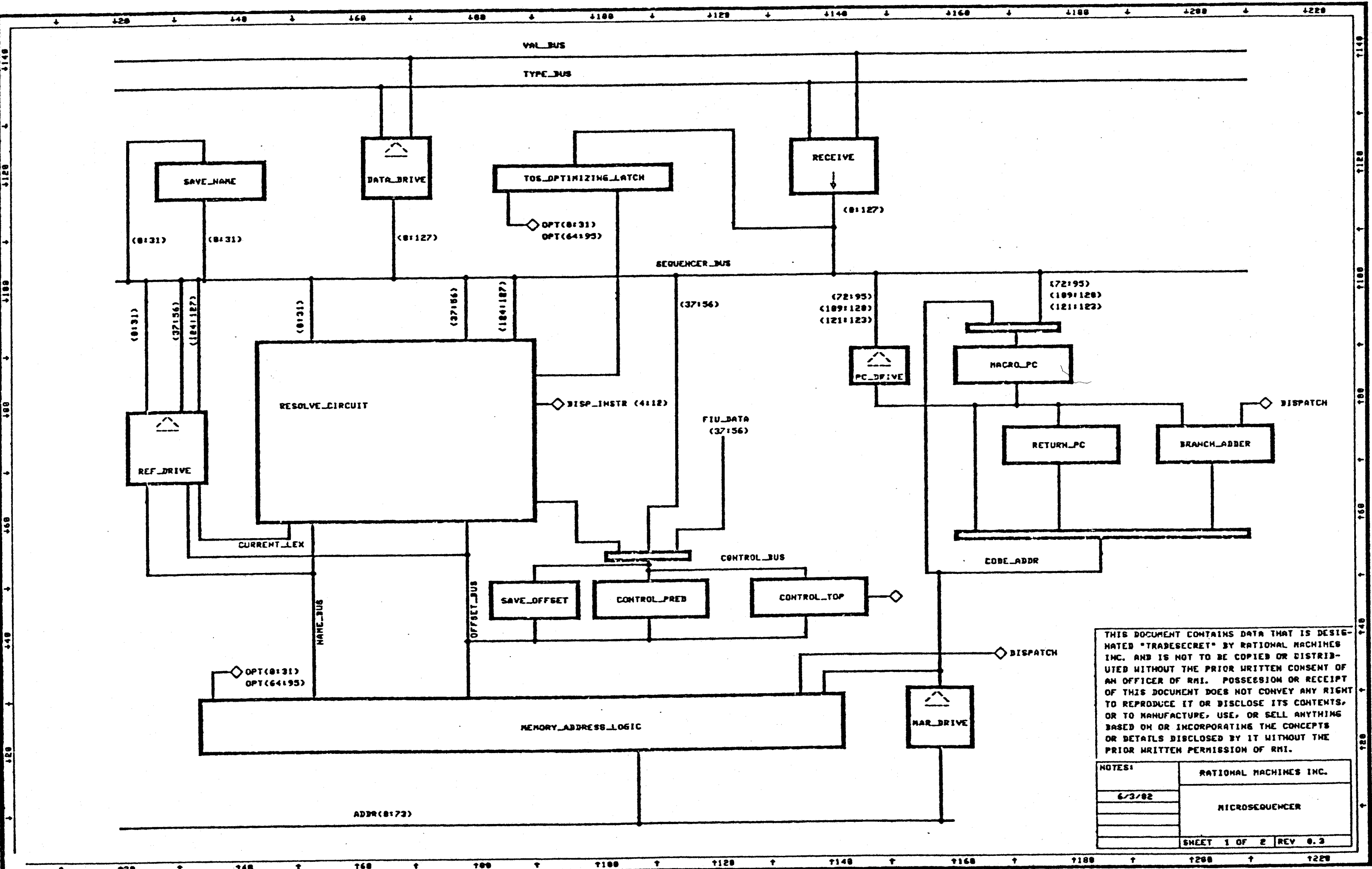
Please return any comments to RJB.

This draft was created on 1/10/1984.


SYSBUS(0,63)
SYSDIAG(0,1)
TYPE(0,63)
VAL(0,63)
ADR(0,63)
SYSBUS_INF
MICROSEQ
TYPE
VALUE
FIU
MEMORY_2MB
(2,3,OR 4 CARDS
PER PROCESSOR)
FIU(0,63)
CPU_DIAG(0,1)
PROCESSORS
PROCESSOR1
PROCESSOR2
PROCESSOR3
IOADAPTER0
IOADAPTER1
IOADAPTER2
IOADAPTER3
PWRCTRL
OPCTRL
TELLINE 4
CONLINE 4
BLU1 4
BLU2 4
UNIBUS0 128
TEMP_SENSE
BLU1 4
UNIBUS1 128
UNIBUS2 128
UNIBUS3 128
IOSUBSYS1
IOSUBSYS2
IOSUBSYS3
IOSUBSYS0
(MASTER)
CONSOLE
PRINTER
DIAG
MODEM
FRONT
PANEL
THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.
NOTES:
5/6/82
RATIONAL MACHINES INC.
R1000
SHEET 1 OF 1
REV 0.1

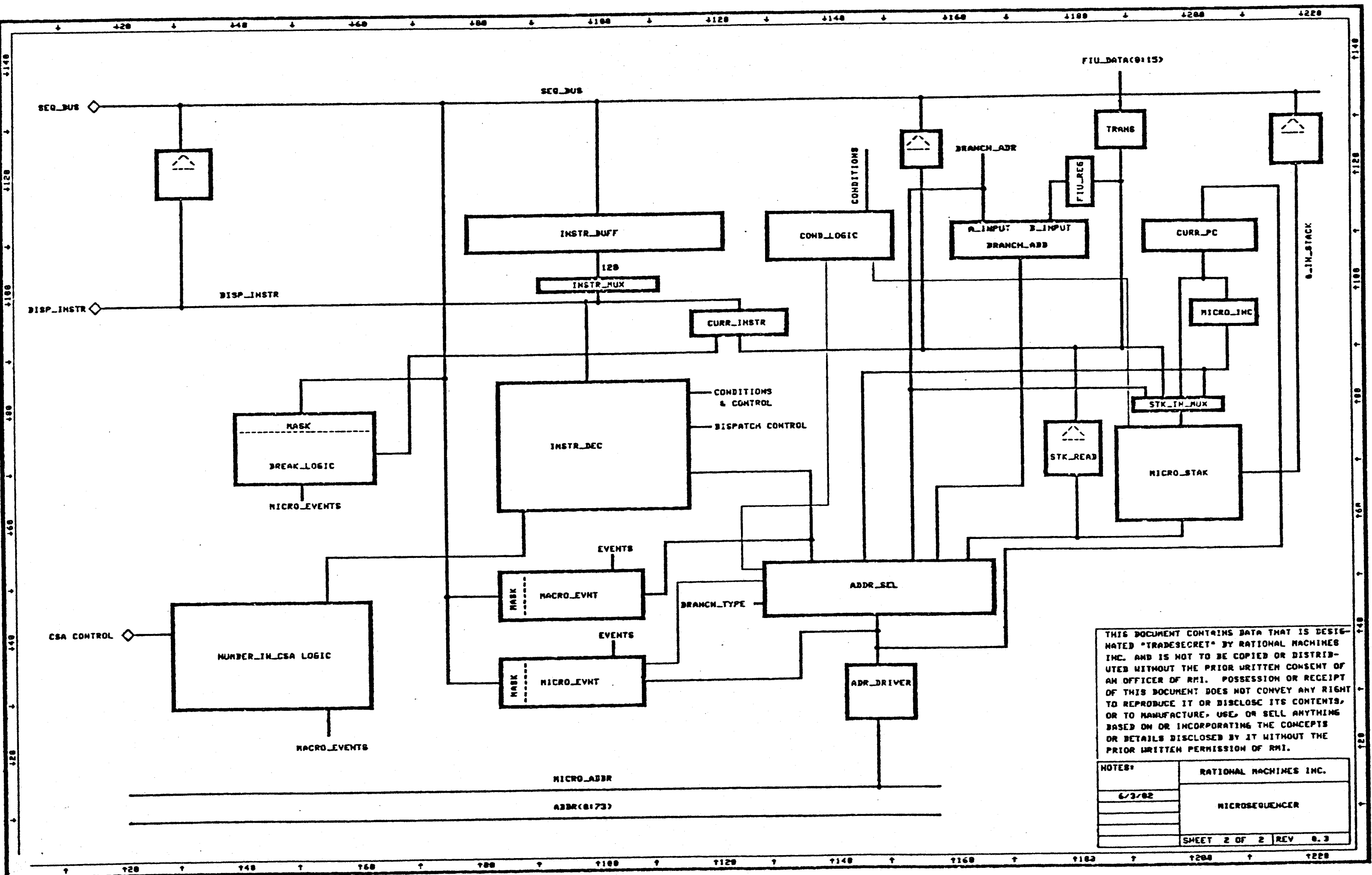

SYSDIAG(0:11)
DIAGNOSTIC CONTROL
DIAGNOSTIC INTERFACE
DIAG_SLU
DBUF_CE
DBUF_WE
DIAG_BUF
ADDR
DATA I/O
DBUF_ADDR (COUNTER)
IODATA(56,63)
IODATA(0,63)
COMLINE 4
TELLINE 4
CON_SLU 4
CONSOLE MUX
P1_CLOCK
PBUF_CE
PBUF_WE
PACKET_BUF
UNI_ADDR(4,14)
BUS_ADDR(0,10)
A_MUX
BUS_ADDR(0,3)
BUS_ADDR(4,10)
P2_CLOCK
BUS_BUF REG
BUS_WWORD COUNTER
NEXT_WWORD
NEW_HEADER
BUF_MUX
IODATA(0,3) IODATA(4,7)
UNIBUS_ADDR(0,17)
A_TERM
U_RCVR
U_A_REG
U_DECODE
BUF_SEL
BOOT_SEL
PWR_SEL
CAL_SEL
UNI_ADDR(0,17)
BUF_CTRL
UNI_INTERRUPT
IO_OUT_REG
BUS_OUT
P2_CLOCK
IO_IN_REG (DIAG SR?)
P1_CLOCK
BUS_IN
OUT_TIME
BID_LOGIC
MODE(0,2)
BID(0,7)
SYSBUS(0,63)
CLOCK
CLOCK_GEN
UNIBUS_DATA(0,15)
D_TERM
U_XCVR
U_IN_REG
U_OUT_REG
U_MUX
IODATA(0,15)
IODATA(16,31)
IODATA(32,47)
IODATA(48,63)
BOOT_LOAD
PWR_SEQ
PWRCTRL
CALENDER
THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.
NOTES: 3/7/82
RATIONAL MACHINES INC.
I/O ADAPTOR
SHEET 1 OF 1
REV 0.2

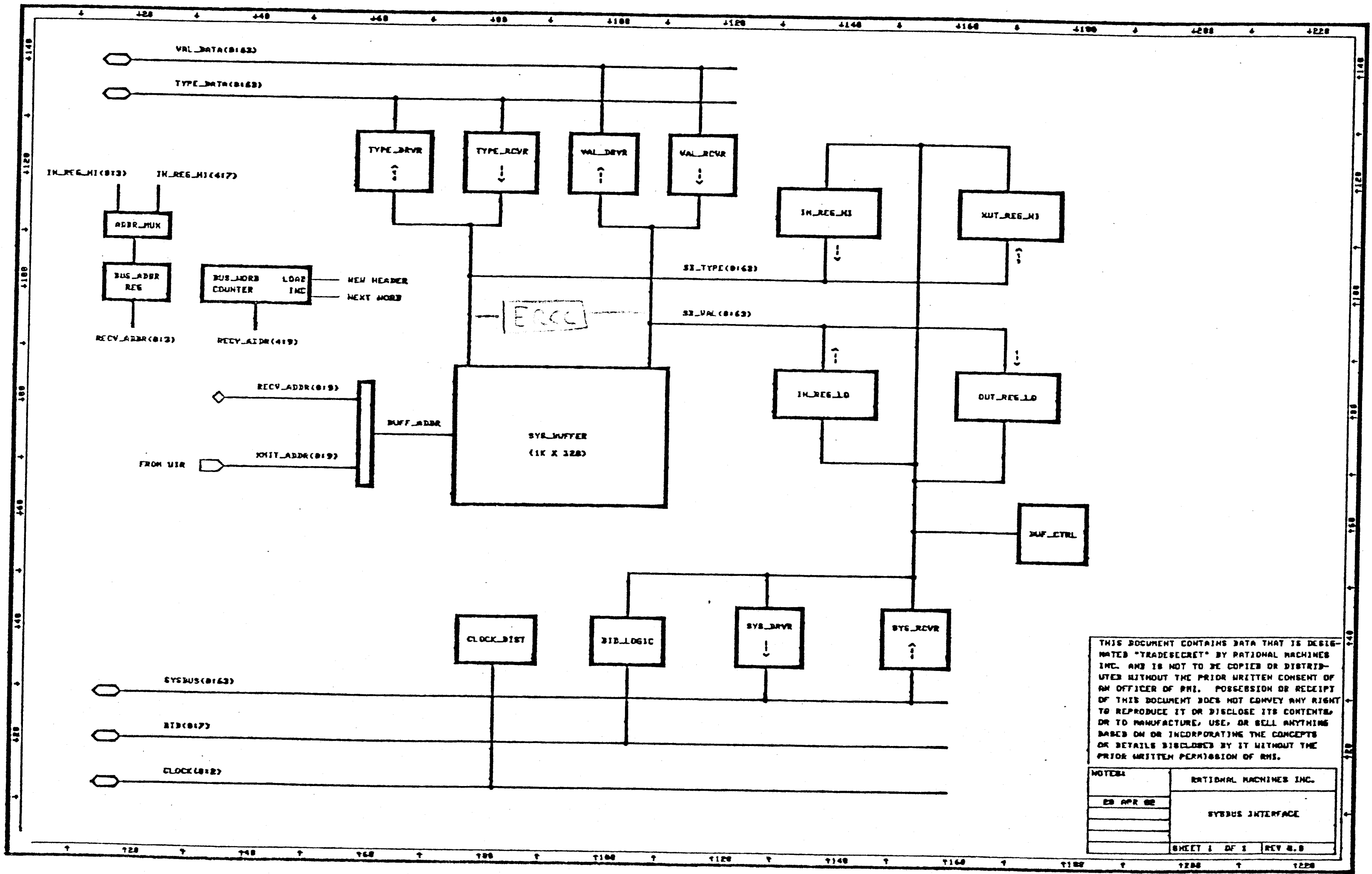

5-INCH 11/24 CHASSIS
11/24 CPU
11/24 LPT_CTRL
256KB MEMORY
DISC_CTRL
TAPE_CTRL
SLU1
SLU2
LINE PRINTER
DISC0
DISC1
DISC2
DISC3
TAPE0
TAPE1
TAPE2
TAPE3
4_SLOT BACKPANEL
ASYNC_AUX
UNIBUS
CS11
FLAT CABLE
EIA_PANEL0 (LINES 0 TO 15)
EIA_PANEL1 (LINES 16 TO 31)
EIA_PANEL2 (LINES 32 TO 47)
EIA_PANEL3 (LINES 48 TO 63)
THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.
NOTES:
4/14/82
RATIONAL MACHINES INC.
I/O SUBSYSTEM
SHEET 1 OF 1
REV 0.0



THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.

NOTES:	RATIONAL MACHINES INC.
6/3/82	MICROSEQUENCER
	SHEET 1 OF 2 REV 0.3


SEQ_BUS
SEQ_BUS
DISP_INSTR
DISP_INSTR
FIU_DATA(0:15)
TRANS
FIU_REG
BRANCH_ADR
A_INPUT
B_INPUT
BRANCH_ADD
CONDITIONS
COND_LOGIC
INSTR_BUFF
128
INSTR_MUX
CURR_INSTR
CURR_PC
MICRO_INC
B_IN_STACK
STK_IN_MUX
STK_READ
MICRO_STAK
MASK
BREAK_LOGIC
MICRO_EVENTS
INSTR_DEC
CONDITIONS
& CONTROL
DISPATCH CONTROL
EVENTS
MASK
MACRO_EVNT
EVENTS
MASK
MICRO_EVNT
ADDR_SEL
BRANCH_TYPE
ADR_DRIVER
CSA CONTROL
NUMBER_IN_CSA LOGIC
MACRO_EVENTS
MICRO_ADDR
ADDR(0:7)
THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.
NOTES:
6/3/82
RATIONAL MACHINES INC.
MICROSEQUENCER
SHEET 2 OF 2
REV 0.3



THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.

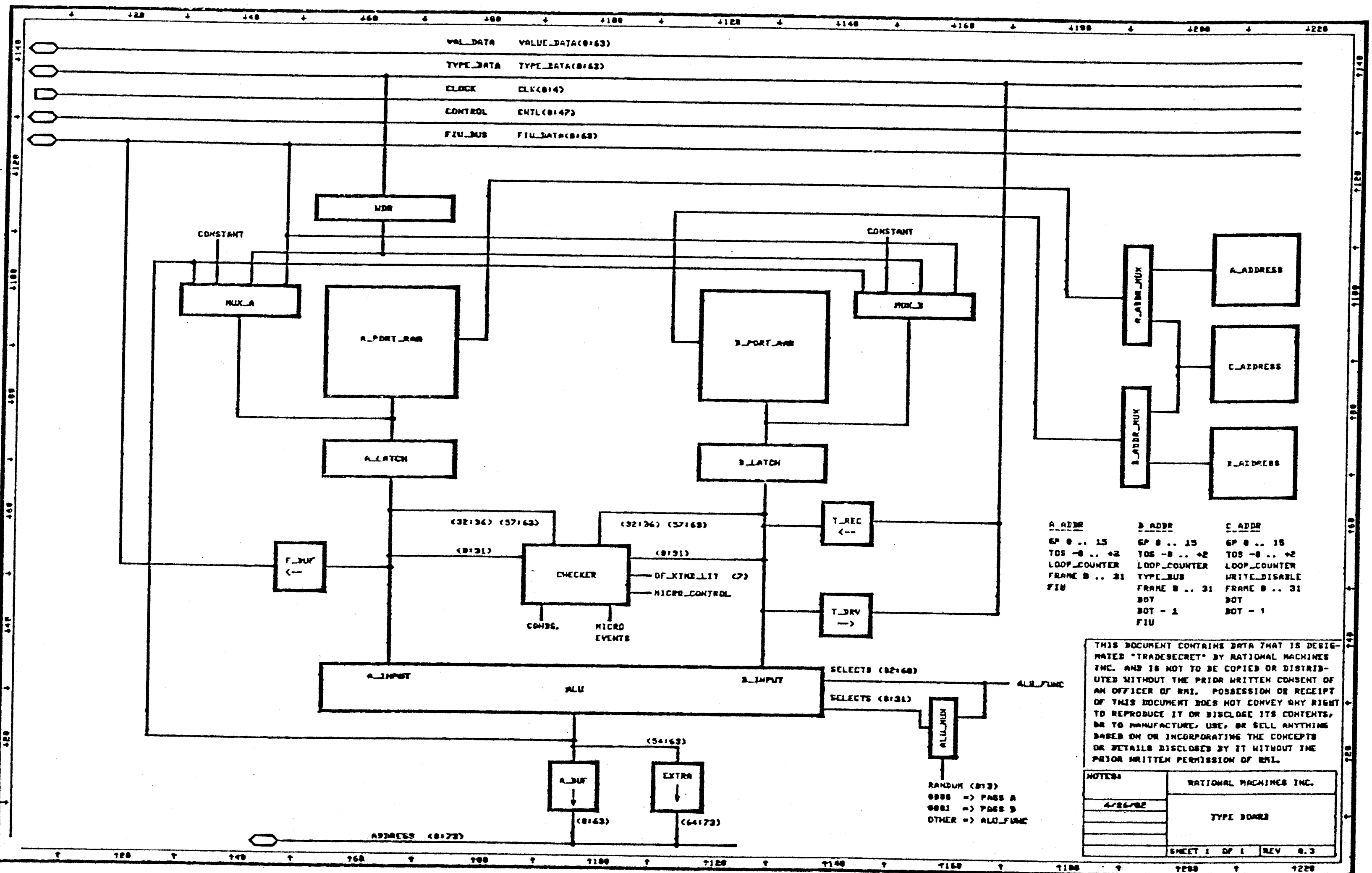
NOTES:

20 APR 82

RATIONAL MACHINES INC.

SYSBUS INTERFACE

SHEET 1 OF 1 | REV 0.0



THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.

NOTES:

4/26/82

RATIONAL MACHINES INC.

TYPE BOARD

SHEET 1 OF 1 REV 0.3

OUTER_FRAME_NAME

A_BUS (0:31)

B_BUS (0:31)

32 BIT COMPARE

32 BIT COMPARE

32 BIT COMPARE

IS_PRIVATE
DERIVED_PRIVACY

IS_PRIVATE
DERIVED_PRIVACY

PRIVACY CHECK LOGIC

A_OP~

BIN_EQ~

BIN_OP~

B_OP~

MICRO_EVENTS

SELECTOR

BIN_EQ
BIN_OP
A_(TOS)_OP
A_(TOS-1)_OP
B_(TOS)_OP
B_(TOS_1)_OP

MICRO_CODE CONTROL

BIN_EQ
BIN_OP
A_(TOS)_OP
A_(TOS-1)_OP
B_(TOS)_OP
B_(TOS_1)_OP
PASS_PRIVACY_CHECK
NOP

MICRO_LIT

A_BUS(57:63)

B_BUS(57:63)

VAR_BIT_COMPARE

VAR_BIT_COMPARE

VAR_BIT_COMPARE

(A,LIT)

(A,B,LIT)

(A,B)

(B,LIT)

MICRO_EVENT

SELECTOR

CLASS_ERROR

CLASS CHECK

MICRO_CODE CONTROL

(A,LIT)
(B,LIT)
(A,B)
(A,B,LIT)
NOP

A_BUS CONDITIONS

HARDWIRED
TEST
CONDITIONS

A_BUS
(57:63)
(32:36)

B_BUS
(57:63)
(32:36)

A_BUS
CONDS

B_BUS
CONDS

TESTABLE CONDITIONS

B_BUS CONDITIONS

VALUE_VAR OR STRUCTURE_VAR
VALUE_VAR
TYPED_VAR
INDIRECT_VAR
STRUCTURE_VAR
IS_MARKED
IN_UTILITY OR HAS_CHILDREN

THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.

NOTES:

4/23/82

RATIONAL MACHINES INC.

TYPE_CHECKER

SHEET 1 OF 1 | REV 0.3

VALUE_DATA (0:63)

FIU_BUS (0:63)

RDR

F_REC

L P R

SHIFT_MUX

LOOP_CNTR

ZERO_DTECT

A ADDR
GP 0 .. 15
TOS -8 .. +2
REG(LOOP_COUNTER)
FRAME 0 .. 31
PRODUCT
ZERO_DTECT
LOOP_COUNTER

B ADDR
GP 0 .. 15
TOS -8 .. +2
REG(LOOP_COUNTER)
FRAME 0 .. 31
VAL_BUS
BOT
BOT - 1

C ADDR
GP 0 .. 15
TOS -8 .. +2
REG(LOOP_COUNTER)
FRAME 0 .. 31
WRITE_DISABLE
BOT
BOT - 1
LOOP_COUNTER

REG_FILE

A_ADDRESS (0:9)

B_ADDRESS (0:9)

C_ADDRESS (0:9)

RAM_ADDR

F_DRV

MULTIPLIER (16 BY 16)

V_XCVR

A_INPUT

ALU

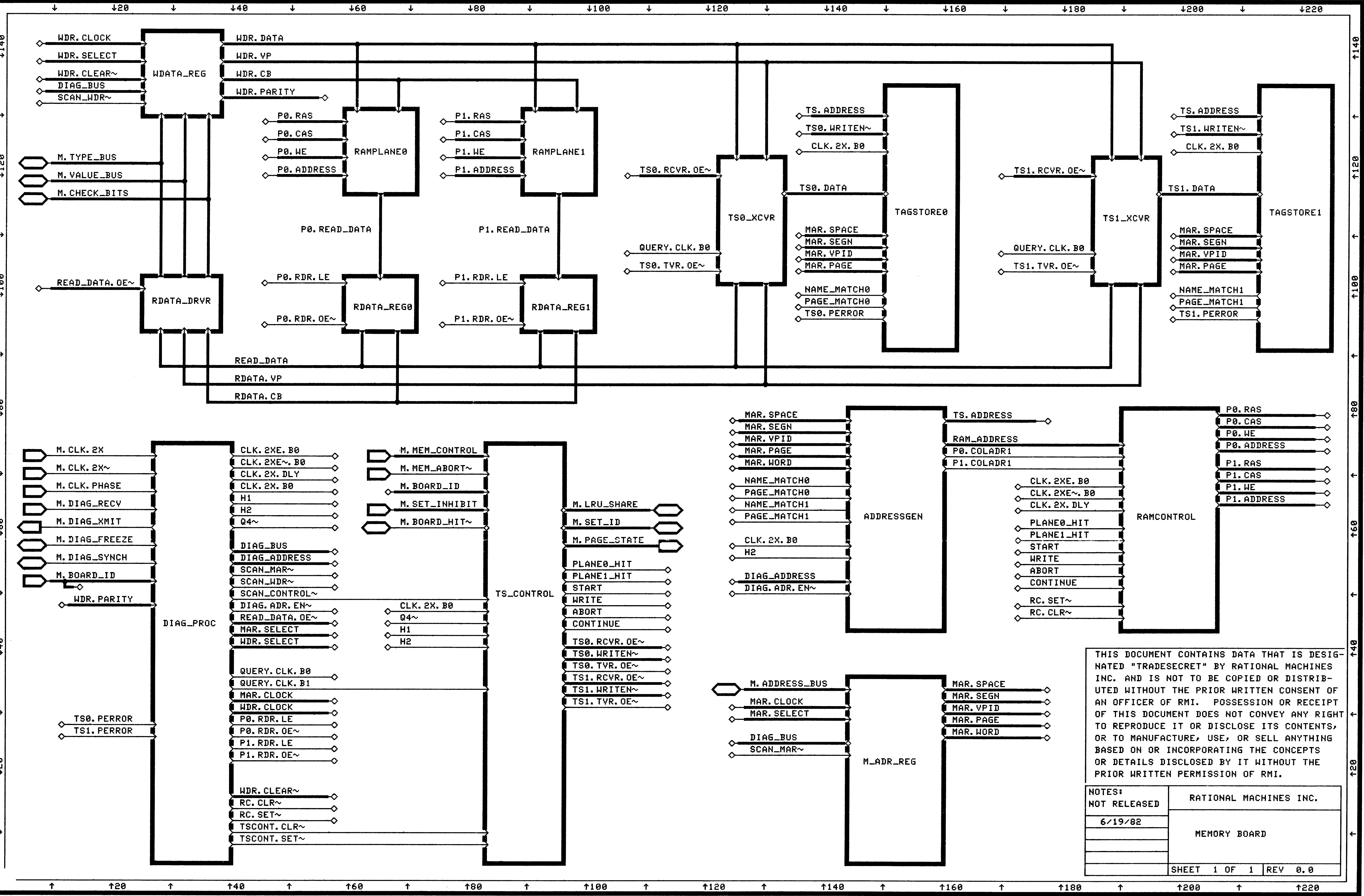
B_INPUT

A_DRV

ADDRESS (0:63)

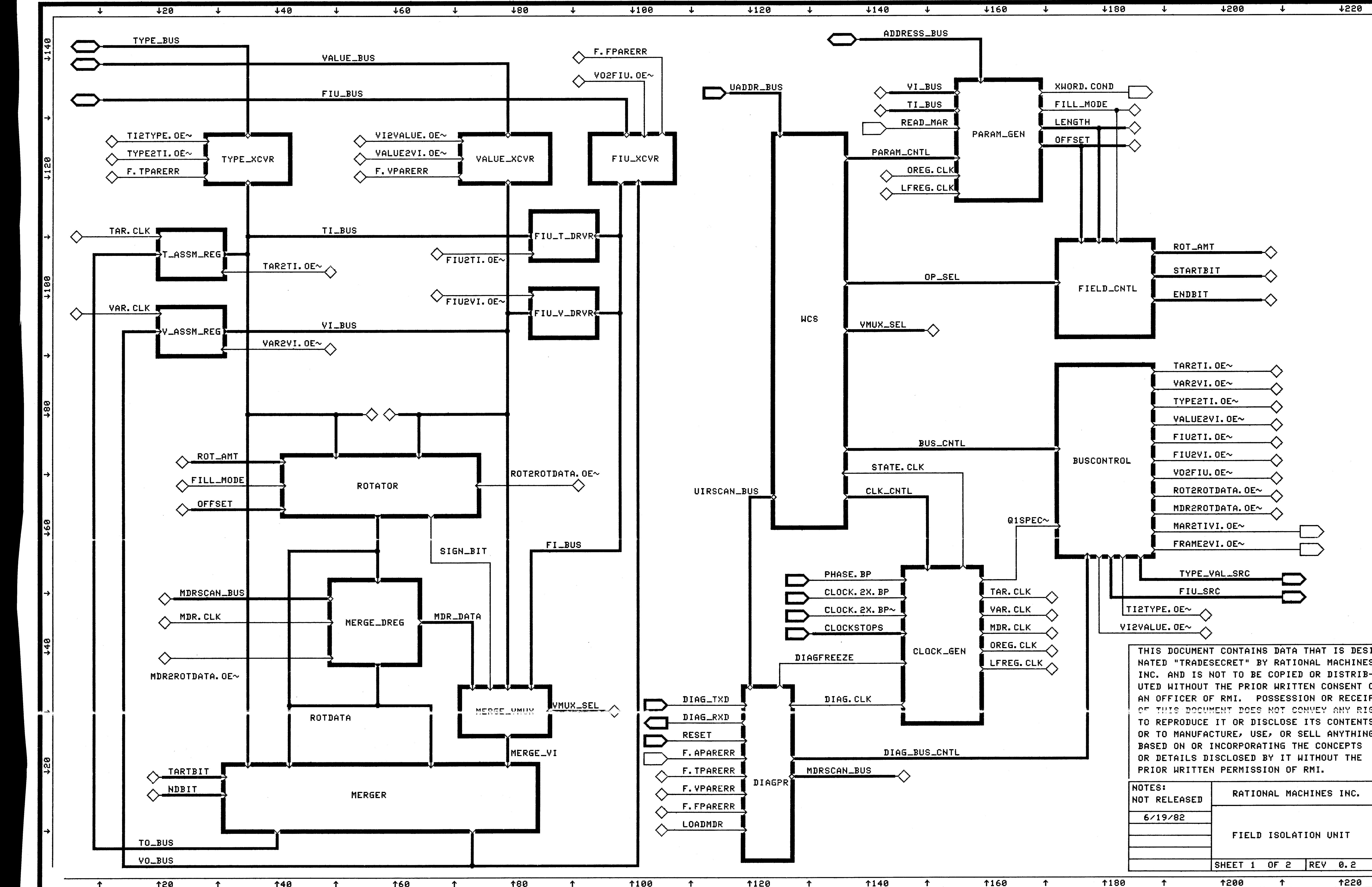
THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.

NOTES:	RATIONAL MACHINES INC.
12 MAY 82	VALUE BOARD
	SHEET 1 OF 1 REV 0.2



THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.

NOTES: NOT RELEASED	RATIONAL MACHINES INC.
6/19/82	MEMORY BOARD
	SHEET 1 OF 1	REV 0.0



THIS DOCUMENT CONTAINS DATA THAT IS DESIGNATED "TRADESECRET" BY RATIONAL MACHINES INC. AND IS NOT TO BE COPIED OR DISTRIBUTED WITHOUT THE PRIOR WRITTEN CONSENT OF AN OFFICER OF RMI. POSSESSION OR RECEIPT OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHT TO REPRODUCE IT OR DISCLOSE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING BASED ON OR INCORPORATING THE CONCEPTS OR DETAILS DISCLOSED BY IT WITHOUT THE PRIOR WRITTEN PERMISSION OF RMI.