K Technical Manual

May 15, 2006

Prepared by James Rauen for GigaMos Systems, Inc.

Portions of this document were either written by or are based on technical notes by Kent Hoult.

Chapter 1

Introduction

This is a preliminary draft of the K processor technical manual. It is intended to describe the K processor from a system software programmer's perspective. It does not dwell on details of the various hardware implementations.

For a full hardware specification of the processor, consult the *Falcon Processor Design Specification* by Kent Hoult and the actual board and chip schematics.

Please observe that any documentation or information about the K processor (including this manual) is proprietary information belonging to GigaMos Systems, Inc.

`James Rauen`
`Consultant`
`GigaMos Systems, Inc.`

Chapter 2

Editor's Note

Revisions have been made to the following sections since the May 27th draft:

4.1 (and ff.) Clocks renamed to CPROC1, CPROC2, CMEM1, CMEM2.

4.3.1.3, 4.3.2 Enforced read wait after a write instruction clarified.

5.3.3 Reference to macro-link bit clarified, "Opcode" column header renamed to "Instruction Format"

5.3.6 DT-HAIRY-NUMBER explained

5.4.2 Caption for ALU Immediate Instruction clarified

8.1 Reference to "yellow alert" trap clarified.

12. ALU Opcodes documentation changes (SIGN, Q Register use)

Future versions of this draft will incorporate further revisions to this document.

—David Saslav

Chapter 3

Notation Conventions

This chapter summarizes the notation conventions used in the manual.

3.1 Numbers

Hexadecimal quantities are preceded by "#x".

The contents of bit fields are always written in binary, without any qualifier.

All other numerical quantities are written in base ten.

3.2 Bit Fields

The bits of an N-bit word are numbered from 0 (the least significant bit) to $N - 1$ (the most significant bit).

Bit fields are written in the form MSB:LSB, where MSB (written in base ten) is the number of the most significant bit in the field and LSB (also written in base ten) is the number of the least significant bit in the field.

Chapter 4

Overview of the K Architecture

4.1 History

The K machine is a high-speed processor heavily optimized to run Lisp. It was originally designed at LMI (Lisp Machine, Incorporated) to run on two NuBus or VME-bus boards. One board is devoted to processor functions, and the other board is devoted to the on-board memory system. The NuBus boards have undergone several small design modifications and are now in use at GSI.

A second design, placing the processor and memory system on chips instead of boards, is currently taking place at GigaMos and SilicArt. The chip design differs from the board design in several ways. Some of the differences are due to the limited amount of space on the chips; several of the boards' features have been scaled down for this reason.

4.2 Board Set vs. Chip Set

The primary intention of this manual is to document the board set. Most of the hardware references will specifically mention chips and signals which appear on the processor board and memory board design sheets. Nevertheless, most of the information in this manual should still be applicable to the chip set design.

Here is a list (by no means exhaustive) of some of the differences between the board and chip designs:

- The size of register frame memory has been reduced for the chip set. There are 64 register frames in the chip set, compared to 256 in the board set.
- The arrangement of bits in the Processor Control, Processor Status, Memory Control and Memory Status registers has been changed.
- The instruction cache has been completely redesigned. The board set has two fully associative sets and a special low core cache.
- The floating point ALUs have been eliminated from the chip set.

4.3 Processor Board

The processor board contains most of the actual processor hardware. This includes the instruction cache, the register memory, the call hardware, and the ALUs. Each of these features is documented in subsequent chapters.

4.4 Memory Board

The memory board contains the processor's on-board memory. It also contains hardware to help implement the virtual memory system, a volatility-based garbage collector, the transporter, and traps.

Part I

Processor Board Hardware

Chapter 5

Timing

This chapter describes the K processor's clocks and pipeline stages.

5.1 Clocks

The K processor uses four clocks for timing. Two of the clocks run at the instruction rate, and the other two run at twice the instruction rate. Two of the clocks can be stopped by the clock generator during memory waits, instruction cache misses, and traps. The other two clocks run the memory system and never stop. They will be referred to as follows:

```
               1X        2X
               ----      ----
Processor -    CPROC1    CPROC2
Memory    -    CMEM1     CMEM2
```

The phasing of the clocks is such that if the CPROC1 clock rises, then the other three will rise at the same time. When the processor clocks stop, they will always stop in the state with both of them low. Whenever the processor clocks stop, it will always be for an integral number of 1X clock ticks. The following timing diagram shows the relative timing:

```
        --    --    --    --                ----- 
CPROC2_|  |__|  |__|  |__|  |______________|     |_

        -----       -----                   ----- 
CPROC1_|     |_____|     |_________________|     |_

        --    --    --    --    --    --    --    -
CMEM2 _|  |__|  |__|  |__|  |__|  |__|  |__|  |__|

        -----       -----       -----       ----- 
CMEM1 _|     |_____|     |_____|     |_____|     |_
```

The nominal time for the current K processor 1X clock is 80 nanoseconds. All four clocks are produced by the clock generator, which is a synchronous state machine running on a 20 nanosecond clock.

The majority of the processor runs on the 1X clocks. Only a few sections need the 2X times. The main uses are producing write enables to RAMs during the first half of a clock tick and controlling the direction of the MFIO bus.

5.2 Pipeline

The K is a pipelined architecture. Instructions normally go through four stages of execution: PC, IR, ALU, and OUTPUT. Some functional I/O ports act like virtual fourth or even fifth stages of the pipeline. The only places where these extra stages show up are in special OS internal routines for modifying special registers, or loading/unloading the call hardware. In fact, for normal instruction execution, only three of the four pipeline stages are visible.

5.2.1 Straight Line Execution

When executing a linear sequence of instructions, the pattern of execution is as follows:

PC A program counter is produced from the PC incrementer. This PC is fed to the instruction cache, and the specified instruction is fetched. If the instruction isn't in the cache, then the processor clock is stopped while a line of four instructions are fetched.

IR The previously fetched instruction is loaded into the instruction register for decoding. Any specified registers are accessed from the register RAM using the current call hardware O, A, and R registers. The accessed left and right sources are stored into the left and right data registers at the end of the clock tick.

ALU The ALU gets the data from left and right registers and an opcode from some IR bits that have been delayed by one clock tick. The ALU output is loaded into the OREG at the end of this clock tick.

OUTPUT The OREG is written back to the register RAM during this tick. The write happens early in the cycle, so the current IR cycle can read the results from the RAM. The OREG data is also driven onto the MFIO bus for functional destinations.

5.2.2 Unconditional Branches and Jumps

Unconditional jumps and branches work by having the current IR bits select the low IR bits as the PC source. The pipeline does exactly the right thing for this case, and the next instruction loaded into the IR will be the one jumped to.

5.2.3 Conditional Branches and Jumps

Conditional transfers are identical to the unconditional ones except that the jump bit is used by the PC mux to control whether the IR or the PC incrementer is used for the next address. The computation of the jump bit shows the pipeline effect quite clearly. In order to test something and then do a conditional branch on it, a three instruction sequence is required. However, for the second and third instructions the ALU is free and may be used for other computations if the compiler can schedule them in.

The first instruction will do an ALU op that produces some status to be branched on. The second instruction will use its branch condition field to select the condition. And the third instruction will be the actual conditional branch. The IR stage must contain the conditional branch instruction at just the time when the jump bit is at the OREG (effectively). Getting the compiler to make use of the extra instructions can triple the machine performance in branch intensive sections of code (such as COND handling).

5.2.4 Dispatches

These show a two stage offset in the pipe similar to conditional branches. A computed PC must be in the OREG at the same time as the NEXT-PC-DISPATCH field is in the IR. This occurs when the dispatch occurs two instructions after the PC calculation.

5.2.5 Subroutine Calls

The timing for a subroutine call (all types) is identical to that of an unconditional jump. The only difference is that the call hardware saves the OPEN and ACTIVE frame pointers, the current PC plus one, and the return destination on the call stack when the instruction reaches the ALU phase.

5.2.6 Subroutine Returns

A subroutine return timing is the same as for unconditional jumps. When the instruction is in its IR phase, the next PC selected is from the call stack. There is a special restriction in the current call hardware implementation: two return instructions cannot be executed consecutively. (The call stack needs one extra tick to move its pointer, read the next entry, and have the next return PC to the PC multiplexer early enough to use for the instruction cache). This is normally handled by the compiler by inserting a NOP between a CALL and a RETURN in the few cases where this occurs and a TAIL-CALL could not be used.

5.3 Functional I/O

Functional I/O occurs to and from registers on the MFIO bus. The bus timing is as follows during normal cycles:

```
          ------------------------                          --
CPROC1 __|                        |________________________|
       ---- ------------------------ ------------------------ -
MFIO       X Output Data from OREG  X  Input Data from FSRC  X
       ---- ------------------------ ------------------------ -
```

This timing is altered during instruction cache misses to get the PC to the memory system.

5.3.1 Functional Destinations

The output data is normally held on the inputs to functional destinations (by a latch) until after the CPROC1 clock rises. This means that functional destinations aren't loaded until the end of the output stage (with certain special exceptions).

VMA (with and without memory start)

The VMA is a transparent latch that will be made transparent during the output phase on the MFIO when it is a destination. This will let it feed the memory map with an address during the output phase. By the time the CPROC1 clock edge rises, the map will have produced a physical address, it will have propagated through the DRAM address buffers, and it will be setting up on the DRAM address lines. This will let the RAS signal to the RAMs go active right after the clock edge (if a this was a start cycle), and the memory system decides the cycle should start now.

RAMs

The various RAMs that are destinations will be written during the output phase of the MFIO bus. These include the Garbage Collector RAM, the Transporter RAM, the Call Stack RAMs, the Heap RAM, the Datatype RAM, and the RETURN destination which will select a location in the register RAM.

The OPEN-ACTIVE-RETURN Destination

This destination actually loads a set of registers that will be loaded into OPEN, ACTIVE, and RETURN at the end of the next clock tick. This extra step means that these registers take one tick longer than most other functional destinations to load. Therefore, the modified frame pointers should not be used to read data until at least five ticks after the write instruction (for all CH- operations other than CH-NOOP). CH-NOOP and all functional sources require only four ticks between read and write operations.

After modifying the OPEN-ACTIVE-RETURN destination, four clock cycles are required before using the O, A, or R register frames or using the OPEN-ACTIVE-RETURN functional source. Five clock cycles are required before performing any open, call, or return operations.

```
Modifying the OPEN-ACTIVE-RETURN Functional Destination
<write OPEN-ACTIVE-RETURN>
<NOP>
<NOP>
<NOP>
<Use O, A, or R register frames>
<perform open, call, or return operations>
```

5.3.2 Functional Sources

Functional sources are decoded during the IR phase of an instruction, and enabled onto the MFIO bus during the FSRC phase of a cycle. They are clocked into the RIGHT register at the end of this cycle.

In general, a functional source that is read/write should not be read until at least 4 ticks after the register was written. This time may differ for functional sources that are slower than normal, such as OPEN-ACTIVE-RETURN.

```
Reading a Functional Source
<write functional_source>
<NOP>
<NOP>
<NOP>
<read functional_source>
```

A reference to a memory system functional source will cause the processor clocks to stop until an active cycle is complete. This blocking will become effective at the beginning of the output phase of an instruction with a memory start destination. A memory source should not be referenced by the instruction immediately after one that starts a cycle. The extra tick will give the memory time to lock if necessary. Memory destinations may be consecutive, because they follow the natural pipeline sequencing.

Chapter 6

Instruction Set

This chapter describes the instruction set of the K processor.

6.1 Introduction

A K machine instruction is 64 bits long. Machine instructions are stored in memory with the word containing the low 32 bits occurring at an even numbered address, followed by the word containing the high 32 bits.

A single instruction is capable of performing several different actions simultaneously. For example, an instruction might do an ALU operation and invoke a call-hardware operation at the same time, or it might do a conditional jump and a register move at the same time. The meaning of an instruction, therefore, can be rather complicated.

6.2 Instruction Register (IR)

The Instruction Register is a 64-bit registered multiplexer which contains the current instruction being executed. For details about how the IR is loaded, see the Instruction Cache chapter.

6.3 Bit Fields

This section describes the different bit fields in K machine instructions. Not all bit fields are used by all instructions, and some bit fields occur at different places in different kinds of instructions. The next section describes how the bit fields fit together to form valid instructions.

The bit fields listed in the *Value* columns of charts in this section are given from most significant bit to least significant bit. A bit is "set" if its value is 1 and "reset" or "not set" if its value is 0. The symbols listed in *Abbreviation* columns are symbols recognized by the assembler.

6.3.1 Status Bit (Bit 63)

If this bit is set and bits 3:1 of the Memory Control Register are 010, executing the instruction causes the statistics counter (a 32-bit counter) to be incremented.

6.3.2 Instruction Trap Bit (Bit 62)

This bit, if set, will cause a trap to occur before the instruction begins to execute. (To the trap handler, it will appear that the instruction is about to execute, but hasn't yet.) This feature is used to implement fast dynamic linking (see the chapter on linking). It is also used by the garbage collector to identify code in oldspace.

6.3.3 X-16 Bit (Bit 61)

The X-16 bit is only used during dispatch operations. When X-16 bit is set in a dispatch instruction, the bottom four bits of the dispatch address are zeroed.

In branch and jump instructions, bit 61 determines whether or not the branch/jump is conditional or unconditional. If bit 61 is zero, the branch/jump is conditional. If bit 61 is one, the branch/jump is unconditional.

6.3.4 Instruction Format Field (Bits 60 to 58)

These three bits, along with the Next PC field (bits 57 to 56) and the Call Hardware Operation field (bits 50 to 48), determine the type of the instruction. The three fields are decoded by PALs to generate the control signals required for the instruction. The instruction types are detailed in the Instructions section below.

Instruction Format	*Next PC*	*CH Op*	Instruction Type
12-bit Instructions:			
X00	00	x0x	Branch Instruction (no 100 call hw op)
X00	00	x1x	Call-Z Instruction
X00	yy	x0x	ALU Instruction (yy not 00)
X00	01	x1x	Call-Dispatch Instruction
18-bit Instructions:			
X01	xx	xxx	ALU Immediate Instruction
24-bit Instructions:			
010	00	x0x	Jump Instruction (no 100 call hw op)
010	00	x1x	Call Instruction
32-bit Instructions:			
011	xx	xxx	32-bit Immediate Instruction
110	xx	xxx	Floating Point ALU Instruction
111	xx	xxx	Floating Point Multiplier Instruction

For Instruction Format fields ending in 00 or 01 above, bit 60 is used for the macro-carry bit. The macro-link bit is ALU-boxed.

The branch and jump instructions cannot be used with the RETURN Call Hardware Operations (call hardware with instruction format 100).

6.3.5 Next PC (Bits 57 to 56)

This field tells where to get the new Program Counter from.

Value	*Abbreviation*	*Meaning*
00		Bits 23 to 0 of the IR
01	NEXT-PC-DISPATCH	OREG (the ALU output)
10	NEXT-PC-RETURN	Call Stack Return PC
11	NEXT-PC-PC+1	PC + 1 (increment the PC)

6.3.6 Boxedness of ALU Output (Bits 55 to 54)

The exact meaning of this field depends on what destination the ALU output is being sent to.

When the destination is a register, this field designates how the box bit of the ALU output will be calculated. This calculation also depends on the Box Mux bit in the Processor Control Register. For normal operation, the Box Mux bit is zero. The other mode (Box Mux bit = 1) is used by the call hardware dump/restore software when it is reloading register frames.

When the destination is a functional destination in the memory system, this field indicates the desired box bits of the MD and VMA registers. For details, see the Transporter RAM chapter.

Other functional destinations don't care about the box bit; in these cases, the boxedness field is ignored.

Register Destinations - Normal Mode

This table shows how the ALU box bit is calculated during normal operation.

Value	*Abbreviation*	*Meaning*
00	BOXED-LEFT	Use box bit of the left source
01	BOXED-RIGHT	Use box bit of the right source
10	UNBOXED	Make the output be unboxed
11	BOXED	Make the output be boxed

Register Destinations - Reload Mode

This table shows how the ALU box bit is calculated during reload operations.

Value	*Abbreviation*	*Meaning*
00	OUTREG0	Use bit 0 of the ALU output
01	none	Make the output be unboxed
10	UNBOXED	Make the output be unboxed
11	BOXED	Make the output be boxed

Memory System Destinations

When writing to the VMA, any of the VMA-START-READ, or any of the VMA-START-WRITE functional destinations, bit 54 of the IR is taken as the VMA box bit.

When writing to the MD, any of the VMA-START-READ, or any of the MD-START-WRITE functional destinations, bit 55 of the IR is taken as the MD box bit.

6.3.7 Data Type Check (Bits 53 to 51)

This field determines what check, if any, should be made to the data types of the ALU inputs. If the ALU inputs fail this test, a datatype trap is caused.

These data-type check cases are not hardwired into the processor; they are downloaded at boot time. It is possible to change the data type checks by changing the contents of the Datatype RAM. But you probably don't want to. The current definitions reside in K-SYS: K; FIRM-DEFINITIONS LISP.

Value	*Abbrev*	*Another Abbreviation*
000	DT-0	DT-NONE
001	DT-1	*spare*
010	DT-2	DT-HAIRY-NUMBER
011	DT-3	DT-BOTH-CHARACTER
100	DT-4	DT-RIGHT-ARRAY-AND-LEFT-STRUCTURE
101	DT-5	DT-RIGHT-LIST
110	DT-6	DT-BOTH-FIXNUM
111	DT-7	DT-BOTH-FIXNUM-WITH-OVERFLOW

Some details:

A DT-HAIRY-NUMBER is a number of any type other than DT-BOTH-FIXNUM and DT-BOTH-FIXNUM-WITH-OVERFLOW.

In the DT-BOTH-CHARACTER case, a trap is caused unless the data type of both the ALU inputs is $$DTP-CHARACTER.

In the DT-RIGHT-ARRAY-AND-LEFT-STRUCTURE case, a trap is caused unless the data type of the left ALU input is $$DTP-STRUCTURE and the data type of the right ALU input is $$DTP-ARRAY (and similarly for DT-RIGHT-LIST and DT-BOTH-FIXNUM).

In the DT-BOTH-FIXNUM-WITH-OVERFLOW case, a trap is caused if either:

1. both ALU inputs do not have $$DTP-FIXNUM, or

2. the ALU operation overflows.

For further details, see the chapter on the Datatype RAM.

6.3.8 Call Hardware Operation (Bits 50 to 48)

This field determines what call-hardware operation the instruction will perform. The eight possibilities are listed below. For further details, see the chapter on the call hardware.

Value	*Abbreviation*	*Meaning*
000	CH-NOOP	No operation
001	CH-OPEN	Open a frame, preparing for a function call
010	CH-CALL	Make a function call
011	CH-OPEN-CALL	Do an OPEN and a CALL
100	CH-RETURN	Return from function call
101	CH-TAIL-OPEN	Open a frame, preparing for tail-recursive call
110	CH-TAIL-CALL	Make a tail-recursive call
111	CH-TAIL-OPEN-CALL	Do a TAIL-OPEN and TAIL-CALL

6.3.9 Destination (Bits 47 to 41)

This field determines where the output of the ALU is stored.

If bit 47 is zero, then output is stored in a register. Bits 46 and 45 tell which frame to use: the current Open frame, the current Active frame, the current Return frame, or the current Global frame. Bits 44 to 41 tell which register in that frame to use. Note that the current Open, Active, and Return frames are determined by the contents of the Open, Active, and Return registers, whereas the current Global frame is determined by the Global Frame Number (bits 40 to 37 of the instruction).

If bit 47 is one, then output is sent to a functional destination. Functional destinations include various special-purpose registers, RAMs, and memory system requests. A list of functional destinations and their encodings appears in the Functional I/O chapter.

Value	*Meaning*
000RRRR	Register RRRR in the Open frame
001RRRR	Register RRRR in the Active frame
010RRRR	Register RRRR in the Return frame
011RRRR	Register RRRR in the Global frame (determined by bits 40 to 37)
1XXXXXX	Functional Destination XXXXXX

6.3.10 Global Frame Number (Bits 40 to 37)

These four bits select the global frame that the instruction uses. Any left-source, right-source, or destination references in this instruction to ”global register RRRR” will use the RRRRth register of this frame. Since there are only four bits with which to select a global frame, the global frames are limited to #x00 to #x0F.

Note that any particular instruction can only access one global frame. To perform an operation on registers in more than one global frame, several instructions must be used.

6.3.11 Jump Condition (Bits 36 to 34)

This field determines a test to perform on the ALU output. If the test succeeds, and the next instruction is a conditional branch instruction, then the branch will occur.

Value	*Test*	*Abbreviations*
000		BR-ALWAYS
001	JINDIR bit	BR-JINDIR
010	Z	BR-EQUAL, BR-ZERO
011	Z	BR-NOT-EQUAL, BR-NOT-ZERO
100	(V XOR N)	BR-NOT-GREATER-OR-EQUAL, BR-LESS-THAN, BR-NEGATIVE
101	(V XNOR N)	BR-GREATER-OR-EQUAL, BR-NOT-LESS-THAN, BR-NOT-NEGATIVE
110	((V XOR N) NOR Z)	BR-GREATER-THAN, BR-NOT-LESS-OR-EQUAL, BR-POSITIVE
111	((V XOR N) OR Z)	BR-LESS-OR-EQUAL

The JINDIR condition means to branch if the JINDIR bit is set. The JINDIR bit is part of the Processor Control Register. It is used when exiting from trap routines while restoring the state of the machine.

6.3.12 Byte Width (Bits 33 to 32)

The two bits in this field are directly wired to the Byte Width inputs of the AMD 29332 ALU. For details about how the Byte Width affects the ALU operations, consult the 29332 documentation. Here is a summary of its effects:

ALU Byte Instructions

Some of the ALU operations (I believe the ones numbered from #x00 to #x5F) are "byte" operations. For these functions, this field determines what size "bytes" the ALU operates on. In most cases, the ALU will treat its inputs as 32-bit quantities. However, in certain cases (fixnum arithmetic, for instance) the ALU performs 24-bit operations on its inputs. In other situations (manipulating characters), the ALU considers only the lowest 8 bits of its inputs.

Value	*Abbrev*	*Meaning*
00	BW-32	32-bit ALU operation
01	BW-8	8-bit ALU operation
10	BW-16	16-bit ALU operation
11	BW-24	24-bit ALU operation

ALU Bit Instructions

The other ALU operations are called "bit" operations. Such operations have two additional arguments, a "shift" field and a "mask" field. For these instructions, the Byte Width field indicates where the shift and mask arguments come from.

If bit 33 is zero, then the shift field is taken from bits 10:5 of the instruction. If bit 33 is one, then the shift field is taken from the ALU's internal shift register. Note that the AMD documentation refers to the shift field as the "position" field.

If bit 32 is zero, then the mask field is taken from bits 4:0 of the instruction. If bit 32 is one, then the mask field is taken from the ALU's internal mask register. Note that the AMD documentation refers to the mask field as the "width" field.

6.3.13 Right Source (Bits 31 to 25)

This field determines where the input to the right side of the ALU comes from.

If bit 31 is zero, then the input comes from a register. Bits 30 and 29 determine which frame the register is taken from: Open, Active, Return, or Global. Bits 28 through 25 determine which register in that frame is used. This pattern is identical to that of the Destination field (bits 47 to 41).

If bit 31 is one, then the input comes from somewhere else in the machine. This is referred to as a "functional source". A table of functional sources and their encodings appears in the Functional I/O chapter. In general, if bits 30 and 29 are both zero, the input comes from somewhere on the processor board. If either bit 30 or bit 29 is one, then the input comes from somewhere on the memory board.

Value	*Meaning*
000RRRR	Register RRRR in the Open frame
001RRRR	Register RRRR in the Active frame
010RRRR	Register RRRR in the Return frame
011RRRR	Register RRRR in the Global frame (determined by bits 40 to 37)
1XXXXXX	Functional Source XXXXXX

6.3.14 Left Source (Bits 24 to 19)

This field determines where the input to the right side of the ALU comes from. The left source cannot come from a functional source; it must come from a register. Bits 24 and 23 determine which frame the register is taken from: Open, Active, Return, or Global. Bits 22 through 19 determine which register in that frame is used.

Value	*Meaning*
00RRRR	Register RRRR in the Open frame
01RRRR	Register RRRR in the Active frame
10RRRR	Register RRRR in the Return frame
11RRRR	Register RRRR in the Global frame (determined by bits 40 to 37)

6.3.15 ALU Opcode (Bits 18 to 12, or 31 to 25)

This field determines the ALU operation that the instruction uses. For details, see the chapters on the ALUs and the ALU Opcodes, and the AMD 29332 documentation.

6.3.16 ALU Shift Field (Bits 10 to 5)

This field determines the "shift" input to the ALU for bit operations. For details, see the chapters on the ALUs and the ALU Opcodes, and the AMD 29332 documentation.

6.3.17 ALU Mask Field (Bits 4 to 0)

This field determines the "mask" input to the ALU for bit operations. For details, see the chapters on the ALUs and the ALU Opcodes, and the AMD 29332 documentation.

6.3.18 Return Destination (Scattered)

This field is used in call instructions. It indicates where the value returned by the call should be placed:

Value	*Meaning*
000RRRR	Register RRRR in the current Open frame
001RRRR	Register RRRR in the current Active frame
010RRRR	Register RRRR in the current Return frame
011RRRR	Register RRRR in the current Global frame (bits 40:37)
10XRRRR	Register RRRR in a newly opened frame
11XRRRR	Register RRRR in a newly tail-opened frame

6.4 Instructions

The instruction set of the K processor is limited to the following combinations of the fields described above.

6.4.1 ALU Instruction

63	62	61	60 58	57 56	55 54	53 51	50 48	47 41	40 37	36 34	33 32
St	Tr	X	Opcode	Next	Box	DTP	Call HW	Destination	Global	Jump	Byte

31 25	24 19	18 12	11	10 5	4 0
Right Source	Left Source	ALU	X	ALU	ALU

This is the most common form of instruction. It can perform an arbitrary ALU operation. The only legal combinations of X-16, Next PC, and Call Hardware Operation are as follows.

X-16	*Next PC*	*Call HW Op*	*Function*
0	11	000	Normal
0	11	001	Open
0	11	101	Tail-Open
0	10	100	Return from subroutine
0	01	000	Dispatch
0	01	001	Dispatch and Open
0	01	101	Dispatch and Tail-Open
1	01	000	Dispatch, zeroing low 4 bits of PC
1	01	001	Dispatch, zeroing low 4 bits of PC, and Open
1	01	101	Dispatch, zeroing low 4 bits of PC, and Tail-Open

"Open" and "Tail-Open" are call hardware operations which allocate new register frames.

subsection ALU Immediate Instruction group

(This instruction is not part of the Rev. 0 prototype version)

tex 63 62 61 60 58 57 56 55 54 53 51 50 48 47 41 40

St	Tr	X	Opcode	Next	Box	DTP	Call HW	Destination	Global	Jump	Byte

31 25	24 19	18	17 0
ALU	Left Source	B	Immediate

end tex end group

6.4.2 32-Bit Immediate Instruction

63	62	61	60 58	57 56	55 54	53 51	50 48	47 41	40 37	36 34	33 32
St	Tr	X	Opcode	Next	Box	DTP	Call HW	Destination	Global	Jump	XX

31 0
Immedate

The call hardware, X-16, and Next PC fields are subject to the same constraints as for the ALU instruction.

6.4.3 Floating Point ALU Instruction

63	62	61	60 58	57 56	55 54	53 51	50 48	47 41	40 37	36 34	33 32
St	Tr	X	Opcode	Next	Box	DTP	Call HW	Destination	Global	Jump	XX

31 25	24 19	18 17	16 9	8 3	2 0
Right Source	Left Source	FPU	Floating Point	FPU	FPU

The call hardware, X-16, and Next PC fields are subject to the same constraints as for the ALU instruction.

6.4.4 Floating Point Multiplier Instruction

63	62	61	60 58	57 56	55 54	53 51	50 48	47 41	40 37	36 34	33 32
St	Tr	X	Opcode	Next	Box	DTP	Call HW	Destination	Global	Jump	XX

31 25	24 19	18 17	16 9	8 3	2 0
Right Source	Left Source	FPU	Floating Point	FPU	FPU

The call hardware, X-16, and Next PC fields are subject to the same constraints as for the ALU instruction.

6.4.5 Conditional Branch Instruction

63	62	61	60 58	57 56	55 54	53 51	50 48	47 41	40 37	36 34	33 32
St	Tr	Co	Opcode	NxPC	Box	DTP	Call HW	Destination	Global	Jump	Byte

31 25	24 19	18 12	11 0
Right Source	Left Source	ALU	Low Bits

This instruction performs a branch. If the Cond field (bit 61) is zero, then the branch is conditional. The branch will take place if the jump condition tested in the previous instruction succeeded. If the Cond field is 1, then the branch is unconditional.

The only legal call hardware operations are 000 (NO-OP), 001 (OPEN), and 101 (TAIL-OPEN). The Next PC field must be 00.

This instruction can also perform any ALU operation which does not require shift and mask fields.

The high twelve bits of the target address are taken from the high twelve bits of PC+1 (the next Program Counter). The low twelve bits are taken from bits 11:0 of the instruction. Therefore, a branch instruction can only branch to other instructions in the same block of 2^{12} instructions.

	23 12	11 0
Target:	Bits 23:12 of PC+1	Bits 11:0 of IR

6.4.6 Call-Z Instruction

63	62	61	60 58	57 56	55 54	53 51	50 48	47 41	40 37	36 34	33 32
St	Tr	D	Opcode	NxPC	Box	DTP	Call HW	Destination	Global	DR	Byte

31 25	24 19	18 12	11 4	3	2 0
Right Source	Left Source	ALU	Address	X	DR

This instruction performs a call into low memory. The return destination is taken from bits 61,36:34,2:0. The Next PC must be 00.

The only legal call hardware operations are 010 (CALL), 011 (OPEN-CALL), 110 (TAIL-CALL), and 111 (TAIL-OPEN-CALL).

This instruction can also perform any ALU operation which does not require shift and mask fields.

The high twelve bits and the low four bits of the target address are zero. The other eight bits come from bits 11:4 of the instruction.

	23 12	11 4	3 0
Target:	0000 00000000	Bits 11:4 of IR	0000

6.4.7 Jump Instruction

63	62	61	60 58	57 56	55 54	53 51	50 48	47 41	40 37	36 34	33 32
St	Tr	Co	Opcode	NxPC	Box	DTP	Call HW	Destination	Global	Jump	Byte

31 25	24	23 0
Right Source	X	Address

This instruction performs a jump. If the Cond field (bit 61) is zero, then the jump is conditional. The branch will take place if the jump condition tested in the previous instruction succeeded. If the Cond field is 1, then the jump is unconditional.

The only legal call hardware operations are 000 (NO-OP), 001 (OPEN), and

101 (TAIL-OPEN). The Next PC field must be 00.

The target address is taken from bits 23:0 of the instruction.

This instruction also performs an ALU operation which passes the Right Source to the Destination.

6.4.8 Call Instruction

63	62	61	60 58	57 56	55 54	53 51	50 48	47 41	40 37	36 32
St	Tr	D	Opcode	NxPC	Box	DTP	Call HW	Destination	Global	DR

31 25	24	23 0
Right Source	D	Address

This instruction performs a subroutine call operation. The call hardware operation (bits 50:48) must be one of 010 (CALL), 011 (OPEN-CALL), 110 (TAIL-CALL), or 111 (TAIL-OPEN-CALL). Bits 61,36:32,24 specify the return destination (where the subroutine is supposed to put its result).

This instruction performs a subroutine call. The return destination is taken from bits 61,36:32,24. The Next PC must be 00.

The only legal call hardware operations are 010 (CALL), 011 (OPEN-CALL), 110 (TAIL-CALL), and 111 (TAIL-OPEN-CALL).

The target address is taken from bits 23:0 of the instruction.

6.4.9 Call-Dispatch Instruction

63	62	61	60 58	57 56	55 54	53 51	50 48	47 41	40 37	36 34	33 32
St	Tr	D	Opcode	NxPC	Box	DTP	Call HW	Destination	Global	DR	Byte

31 25	24 19	18 12	11	10 5	4	3	2 0
Right Source	Left Source	ALU	X	ALU	X	X	DR

This instruction performs a subroutine call to an address computed by the ALU during the last instruction. The return destination is taken from bits 61,36:34,2:0. The Next PC must be 01.

The call hardware operation must be one of 010 (CALL), 011 (OPEN-CALL), 110 (TAIL-CALL), or 111 (TAIL-OPEN-CALL).

The X-16 bit, if set, causes the low four bits of the target address to be zeroed.

This instruction can also perform any ALU operation which does not require a mask field.

Chapter 7

Program Counter

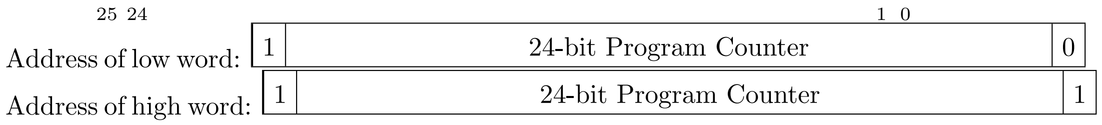
7.1 Introduction

The Program Counter is a 24-bit quantity. It addresses the current instruction being executed. There is no explicit PC register; instead, the PC appears at the output of a 4-way multiplexer. The details are described in the Hardware section below.

7.2 Relation of PC to Virtual Addresses

The 24 bits of PC allow an address space of 2^{24} 64-bit instructions. This works out to 2^{25} 32-bit words, exactly half of virtual memory space. Instructions are only stored in the high half of virtual memory.

Since there are 32 bits at each virtual address, and an instruction is 64 bits wide, each value of the PC must correspond to two virtual addresses. A PC is converted into a pair of virtual addresses by these formulas: Address of low word = $2PC + 2^{25}$; Address of high word = $2PC + 2^{25} + 1$.



7.3 Hardware

This section describes the hardware associated with the Program Counter.

7.3.1 PC Mux

The PC itself appears as the output of this 4-way, 24-bit multiplexer.

The four sources for the multiplexer are as follows. (0) PCINC, the output of the PC incrementer. (1) The Return PC register from the call hardware. (2) OUTREG, the ALU's output register. (3) Bits 23:0 of the Instruction Register.

7.3.2 PC Incrementer (PCINC)

This is a registered 24-bit incrementer.

7.3.3 Delayed PC Incrementer

This is a 24-bit register. Its input lines are connected to PCINC; this effectively delays PCINC for one clock tick.

7.3.4 Old PC Registers

The last two PC's are kept in a pair of registers. These PC's are needed to correctly save and restore the machine state during trap entry/exit.

Chapter 8

Register Memory

This chapter describes the organization of the K processor's general-purpose registers.

8.1 Register Frames

The processor's general purpose registers are organized into 256 *register frames*, each of which contains sixteen 33-bit registers. The contents of each register are arranged according to the formats outlined in the Storage Conventions chapter.

Sixteen of the register frames (those numbered from #x00 to #x0F) are reserved as *global frames*. These frames are not affected by the various function call operations performed by the call hardware. The other 240 *local frames* provide fast storage for local variables, function arguments, and returned values.

8.2 Open, Active, and Return Frame Registers

At any particular time, the processor has access to three of the local frames. These three frames are determined by the contents of three special-purpose 8-bit registers: the Open Frame, Active Frame, and Return Frame registers. The registers in these three frames may be used as sources or destinations for any ALU operation. (See the Instruction Set chapter).

Normally, the contents of the Open Frame, Active Frame, and Return Frame registers are manipulated only by the call hardware. It is possible to directly read and write their contents by accessing functional destination 1001010 (OPEN-ACTIVE-RETURN). The only code which should write this destination is the system software which sets up, dumps, and restores the call hardware.

The assembler provides a shorthand for referring to registers in the current Open, Active, and Return frames. The sixteen registers in the frame addressed by the Open Frame register are labeled O0, O1, and so on up to O15. Likewise, the sixteen registers in the current active frame are labeled A0 through A15. The sixteen registers in the current return frame are labeled R0 through R15.

8.3 Global Registers

The processor always has access to the sixteen global frames. Each instruction contains a Global Frame Number field which designates which global frame the instruction wants to use. The registers in this global frame may be used as sources or destinations for any ALU operation.

8.4 Differences in Chip Set

Due to space limitations, there are only 64 register frames on the processor chip. The high two bits of the Open Frame, Active Frame, and Return Frame registers are ignored. There are still 16 global frames, leaving 48 frames for locals.

Chapter 9

Call Hardware

This chapter describes the K processor's function call hardware.

9.1 Organization

The visible features of the call hardware include the *call stack*, which is used to push and restore machine state during function call operations, and the *free frame heap*, which keeps track of unused register frames.

The call stack is used to save and restore processor state during function call operations. Each entry on the call stack represents, at least conceptually, the Lisp Machine idea of a stack frame. When a function is called, an entry is pushed to save the state of the caller; when a function returns, an entry is popped to re-establish the state of the caller.

The free frame heap keeps track of the 256 register frames. It maintains (in hardware) a list of which frames are currently in use and which frames are currently unused. The heap hardware is also responsible for causing a "yellow alert" trap (by asserting TRAP_STACK_OVF) whenever the call hardware is about to run out of frames.

However, the hardware is *not* responsible for causing a trap when an *underflow* is about to occur. Instead, it is the responsibility of the software to set up the call hardware to regain control on underflow. This is achieved by replacing the Return PC at the base of the call hardware stack with the PC of a special routine. This routine will:

- regain control when all frames have exited;
- read in the top section of the call stack from memory;
- branch to the Return PC originally found at the base of the hardware stack.

9.2 Registers and Memory

9.2.1 Call Stack

The call stack is a stack on the processor board which has 256 entries. Each entry consists of five fields: an Open frame number (8 bits), an Active frame number (8 bits), a return PC (24 bits), a global frame number (4 bits), and a return destination (7 bits). The call stack is implemented with five RAMs (one for each field) which are addressed by an eight-bit Call Stack Pointer.

Call Stack Pointer (CSP)

This is an eight-bit up/down counter. It addresses the five RAMs which comprise the call stack. It can be read or written from bits 7:0 of the MFIO bus by accessing functional destination 1001100.

Call Stack Open RAM

This is a 256 x 8-bit RAM which contains the open-frame portion of each call stack entry. It is addressed by the CSP. The call hardware loads it with the contents of the Previous Open Frame register during OPEN and OPEN-CALL operations. The call hardware loads the Open Frame register with the contents of the Call Stack Open RAM during RETURN operations.

Call Stack Active RAM

This is a 256 x 8-bit RAM which contains the active-frame portion of each call stack entry. It is addressed by the CSP. The call hardware loads it with the contents of the Previous Active Frame register during OPEN and OPEN-CALL operations. The call hardware loads the Active Frame register with the contents of the Call Stack Active RAM during RETURN, RETURN-NEW-OPEN, and RETURN-NEW-TAIL-OPEN operations.

Call Stack Return Destination RAM

This is a 256 x 7-bit RAM which contains the return-destination portion of each call stack entry. It is addressed by the CSP.

The call hardware loads the Return Destination RAM with the Return Destination field of the Instruction Register (IR) during CALL and OPEN-CALL operations. (Depending on the particular instruction, this is either bits 61,36:34,2:0 or bits 61,36:32,24 of the IR). The call hardware reads the Return Destination RAM during RETURN operations.

The Return Destination RAM can also be read or written from the MFIO bus. It appears as bits 30:24 of the MFIO bus when the RETURN-PC-RETURN-DESTINATION (1001011) functional source/destination is accessed.

Call Stack Global Frame Number RAM

This is a 256 x 4-bit RAM which contains the return-immediate portion of each call stack entry. It is addressed by the CSP.

The call hardware loads the Global Frame Number RAM with the Global Frame Number field of the IR during CALL and OPEN-CALL operations. The RAM can also be loaded with the four Miscellaneous bits in the Processor Control Register.

For reasons unknown, this RAM is also occasionally referred to as the "Call Stack Return Immediate" RAM.

Call Stack Return PC RAM

This is a 256 x 24-bit RAM which contains the return-PC portion of each call stack entry. It is addressed by the CSP.

The call hardware loads the Return PC RAM with the contents of the delayed-incremented-PC register during CALL and OPEN-CALL operations.

The processor can also read or write the Return PC RAM by using bits 23:0 of the MFIO bus, accessing the RETURN-PC-RETURN-DESTINATION functional source/destination.

9.2.2 Free Frame Heap

The free frame heap keeps track of which local frames are currently being used and which local frames are available for use. The heap is a 256 x 8 bit RAM organized in the following manner:

```
              +-----------------------+
              |  Extra Local Frames   |  FF
              |    for the dumper     |
              |           .           |
              |           .           |
              |           .           |
              +-----------------------+
              |    (Global Frames)    |
              |           .           |
              |           .           |
              |           .           |
              +-----------------------+
Initial HP -> |  Local Frames in Use  |
              |           .           |
              |           .           |
              |           .           |
              +-----------------------+
        HP -> |  Unused Local Frames  |
              |           .           |
              |           .           |
              |           .           |  00
              +-----------------------+
```

The contents of the heap are always a permutation of the 256 frame numbers (#x00 to #xFF). When the processor is booted, some of the boot code is responsible for setting up the Heap RAM. Each of the 256 entries in the Heap RAM should contain a different value.

Heap Pointer (HP)

This is an eight-bit up/down counter. It is decremented during OPEN, OPEN-CALL, and TAIL-OPEN operations. It is incremented during RETURN and TAIL-CALL operations.

When the HP reaches zero, a ”yellow alert” (”heap empty”) trap is caused. The handler for this trap is responsible for dumping out the contents of register memory and rearranging the call hardware so that there is more available space.

The HP can be read and written by accessing functional source/destination CALL-HP-SP (1011100).

9.2.3 Open, Active, and Return

The (eight-bit) Open, Active, and Return registers are modified during function call operations. These operations are implemented in hardware and are described in one of the sections below. It is possible to directly change the contents of the Open, Active, and Return registers by reading and writing a functional source/destination, but there are very few cases where this is needed.

Open Frame Register (OF)

This is an eight-bit register whose contents identify the current Open frame. It is clock enabled (i.e., a new value is loaded) only when an OPEN, OPEN-CALL, TOPEN, or TOPEN-CALL operation is invoked, or when the OPEN-ACTIVE-RETURN functional destination is written.

The Open register can be loaded from any of the following: (0) The open-frame entry at the top of the call stack, (1) The Previous Open Frame register, (2) The Return Frame register, or (3) The contents of the heap (addressed by HP). The call hardware selects the source depending on the particular call hardware operation.

The Open register can also be read and written from bits 23:16 of the MFIO bus by accessing the OPEN-ACTIVE-RETURN functional source/destination.

The Previous Open Frame register is loaded with the contents of the Open register at each clock tick. It is connected to the call stack open RAM and the Open multiplexer. It is used for delayed writes to the call stack open RAM and for undoing the previous call hardware operation in the event of a trap.

Active Frame Register (AF)

This is an eight-bit register whose contents identify the current Active frame. It is clock enabled only when a call hardware operation needs to modify it, or when the OPEN-ACTIVE-RETURN functional destination is written.

The Active register can be loaded from any of the following: (0) The Open Frame register, (1) The active-frame entry at the top of the call stack, (2) The Return Frame register, or (3) The contents of the heap (addressed by HP). The call hardware selects the source depending on the particular call hardware operation.

The Active register can also be loaded from bits 15:8 of the MFIO bus by accessing the OPEN-ACTIVE-RETURN functional source/destination.

There are two Previous Active Frame registers. Each is loaded with the contents of the Active Frame register each clock tick. The output of one is used for undoing the previous call hardware operation during traps, and the other is used for delaying write data to the call stack RAMs.

Return Frame Register (RF)

This is an eight-bit register whose contents identify the current Return frame. It is clock enabled only when a call hardware operation needs to modify it, or when the OPEN-ACTIVE-RETURN functional destination is written.

The Return register can be loaded from either of the following: (0) The Active Frame register, or (1) The Previous Return Frame register. The call hardware selects the source depending on the particular call hardware operation.

The Return register can also be loaded from bits 7:0 of the MFIO bus by accessing the OPEN-ACTIVE-RETURN functional source/destination.

The Previous Return Frame register is loaded with the contents of the Return Frame register at each clock tick. Its output is connected to the Return register multiplexer.

9.3 Call Hardware Operations

There are eight call hardware operations. One of these operations (RETURN) has three distinct forms, depending on the return destination, so there are actually ten operations available. Each operation is described below.

9.3.1 NO-OP

Has no effect on the call hardware.

9.3.2 OPEN

The call hardware OPEN operation is used in preparation for a function call. It pushes the current Open and Active frame numbers onto the call stack, so that they can be restored after the function returns. It allocates a new Open frame. Subsequent instructions should move the function's arguments into the Open frame registers, and then issue a call hardware CALL operation when the code is ready to call the function.

In detail, these are the effects of a call hardware OPEN operation:

- Increment the call stack pointer (CSP)
- Load the Call Stack Active RAM (addressed by CSP) with the contents of the Active Frame register.
- Load the Call Stack Open RAM (addressed by CSP) with the contents of the Open Frame register.
- Load the Open Frame register with the contents of the Heap RAM (addressed by HP).
- Decrement the heap pointer (HP).

9.3.3 CALL

The CALL operation is used to execute a function call. It must be preceded by a corresponding OPEN operation. It writes a return PC and the instruction's Return Destination field onto the call stack. It moves the contents of Open Frame register into the Active Frame register.

In detail, these are the effects of a call hardware CALL operation:

- Move the contents of the Open Frame register into the Active Frame register.
- Load the Call Stack Return PC RAM (addressed by CSP) with the contents of the PCINC register.

- Load the Call Stack Return Destination RAM (addressed by CSP) with the Return Destination field of the instruction in IR.
- Load the Call Stack Global Frame Number RAM (addressed by CSP) with the Global Frame Number field of the instruction in IR.

9.3.4 OPEN-CALL

The OPEN-CALL operation combines the effects of an OPEN and a CALL operation. It saves time when calling a function with zero or one arguments.

9.3.5 RETURN

There are three different kinds of RETURN operations. They are distinguished by the instruction's Return Destination field:

RDest	*Return Type*	*Where to put returned value*
000RRRR	NORMAL	Register RRRR of the Open frame
001RRRR	NORMAL	Register RRRR of the Active frame
010RRRR	NORMAL	Register RRRR of the Return frame
011RRRR	NORMAL	Register RRRR of the Global frame
10XXXXX	OPEN	Register O0 of a new Open frame
11XXXXX	TOPEN	Register O0 of a new tail-open frame

RETURN (NORMAL)

The RETURN instruction discards the current Return frame by pushing it onto the heap. It moves the contents of the Active Frame register into the Return Frame register. Then it pops the Active Frame, Open Frame, Return PC, and Return Destination off of the call stack. The Return PC is used to fetch the next instruction if the PC multiplexer is set that way (as it usually is for a return instruction). The Return Destination will be delayed until the OUTPUT phase, and used as the destination at that time.

In detail, these are the effects of a normal call hardware RETURN operation:

- Increment the heap pointer (HP).
- Store the contents of the Return Frame register into the Heap RAM (addressed by HP).

- Move the contents of the Active Frame register into the Return Frame register.
- Load the Active Frame register with the contents of the Call Stack Active RAM (addressed by CSP).
- Load the Open Frame register with the contents of the Call Stack Open RAM (addressed by CSP).
- Configure the PC multiplexer to take the next PC from the Call Stack Return PC RAM.
- Read the return destination from the Call Stack Return Destination RAM. Configure the Destination logic to put the ALU output (one clock tick from now) into this destination.
- Decrement the call stack pointer (CSP).

RETURN-NEW-OPEN

This operation combines the effects of a RETURN and an OPEN. The newly opened frame is easy to allocate; instead of discarding the Return frame, as RETURN does, RETURN-NEW-OPEN uses the old Return frame as a new Open frame. The destination in this case will always be one of the registers in the newly opened frame.

In detail, these are the effects of a call hardware RETURN-NEW-OPEN operation:

- Load the Open Frame register with the contents of the Return Frame register.
- Load the Return Frame register with the contents of the Active Frame register.
- Load the Active Frame register with the contents of the Call Stack Active RAM (addressed by CSP).
- Configure the PC multiplexer to take the next PC from the Call Stack Return PC RAM.
- Read the return destination from the Call Stack Return Destination RAM. Configure the Destination logic to put the ALU output (one clock tick from now) into this destination.

RETURN-NEW-TAIL-OPEN

This operation combines the effects of a RETURN and a TAIL-OPEN. The destination in this case will always be one of the registers in the newly tail-opened frame.

In detail, these are the effects of a call hardware RETURN-NEW-TAIL-OPEN operation:

- Load the Open Frame register with the contents of the Return Frame register.
- Load the Return Frame register with the contents of the Active Frame register.
- Load the Active Frame register with the contents of the Call Stack Active RAM (addressed by CSP).
- Configure the PC multiplexer to take the next PC from the Call Stack Return PC RAM.
- Read the return destination from the Call Stack Return Destination RAM. Configure the Destination logic to put the ALU output (one clock tick from now) into this destination.
- Decrement the call stack pointer (CSP).

9.3.6 TAIL-OPEN

The TAIL-OPEN (or TOPEN) operation is used in preparation for a tail-recursive function call. Tail recursive call hardware operations do not affect the call stack; there is no need to save any of the caller's state. All that a TAIL-OPEN operation has to do is obtain a fresh frame from the heap.

In detail, these are the effects of a call hardware TAIL-OPEN operation:

- Load the Open Frame register with the contents of the Heap RAM (addressed by HP).
- Decrement the heap pointer (HP).

9.3.7 TAIL-CALL

The TAIL-CALL (or TCALL) operation executes a tail-recursive function call. It must be preceded by a corresponding TAIL-OPEN operation.

In detail, these are the effects of a call hardware TAIL-CALL operation:

- Increment the heap pointer (HP).
- Store the contents of the Return Frame register into the Heap RAM.
- Load the Return Frame register with the contents of the Active Frame register.
- Load the Active Frame register with the contents of the Open Frame register.

9.3.8 TAIL-OPEN-CALL

The TAIL-OPEN-CALL (or TOPEN-CALL, or TOPEN-TCALL) operation combines the effects of a TAIL-OPEN and a TAIL-CALL.

In detail, these are the effects of a TAIL-OPEN-CALL operation:

- Load the Open and Active Frame registers with the contents of the Return Frame register.
- Load the Return Frame Register with the (previous) contents of the Active Frame register.

Chapter 10

Instruction Cache

The instruction cache is a direct access cache with two distinct sections. The first 8K instructions correspond to PC locations 0 to 8191. This is the low-core cache. Once it has been pre-loaded and enabled, it will never miss. Its purpose is to provide fast execution of commonly used functions. These include the trap entry/exit functions, the CALLZ functions, and whatever others are put here.

The top half of the cache is organized as 2048 lines of 4 instructions. When enabled, this will be the general purpose cache for all code not in low core.

10.1 Cache Hits

If the low-core cache is enabled, then any access to a PC in the range 0 to 8191 will cause a cache hit. This range will cause a miss if the low-core cache is disabled.

In the regular cache section, the high PC bits will be compared against the tag memories. If they match, then a cache hit occurs, and the instruction will come from the cache.

The tag RAM has a reset line that is connected to the normal cache enable bit. When disabled, all of the tag bits are zeroed, making them invalid. When enabled, they are loaded as each line is read in.

10.2 Cache Misses

When the cache misses, a line of data (four instructions) will be read from memory. The normal timing for a cache load from local memory is as follows:

```
         _   _   _   _   _   _   _   _   _   _   _   _   _   _
CMEM1  _| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |_| |

      _                                                      _
CPROC1 |____________________________________________________| |

         __________________________________________________
MISS   _|                                                  |___
MFIO       PC         I0L I0H I1L I1H I2L I2H I3L I3H     FS

       ----------                                    ----------
RAS              |__________________________________|

       --------------                                ----------
CAS                  |______________________________|
```

All of the items shown for the MFIO bus occur during the functional source half of the clock cycle (CMEM1 low). The PC is transferred to the memory chip during the first clock of the load cycle during the low half tick. This is the only time data is transferred to the memory chip during this portion of the clock. The other half tick is still used for the functional destination data as usual.

Chapter 11

ALUs

The processor's main ALU is Advanced Micro Devices' Am29332 Arithmetic Logic Unit. For a complete specification of its behavior, consult the 29332's documentation. A summary of the ALU's opcodes appears in the ALU Opcodes chapter.

An ALU operation is determined by the ALU Opcode and Byte Width fields of an instruction. Bit operations might also require the Shift and Mask fields.

The board set is also designed to use Weitek floating point hardware. However, this design had to be abandoned because the state of the Weitek hardware cannot be saved and restored during trap entry/exit.

The left and right inputs to the ALUs are determined by the Left Source and Right Source fields of an instruction. The ALU output is registered in OREG, the ALU output register.

Chapter 12

Datatype RAM

Chapter 13

ALU Opcodes

This chapter describes each of the 128 opcodes available on the ALU. (Ref AMD manual?)

Several abbreviations are used in the following chart. "L" represents the left source of the ALU. "R" represents the right source of the ALU. "Status" refers to the ALU's internal status register. "Q" refers to the ALU's internal Q register. "Foo:Bar" means the 64-bit quantity whose high 32 bits come from Foo and whose low 32 bits come from Bar. All shift instructions which use the Q register use it as the low word.

A more detailed description of the stranger opcodes appears after the chart.

Value	*Abbreviation*	*Description*
#x00	SETL	Zero extend left source
#x01	SETR	Zero extend right source
#x02	SEX-R	Sign extend left source
#x03	SEX-R	Sign extend right source
#x04	PASS-STATUS	Contents of Status register
#x05	PASS-Q	Contents of Q register
#x06	LOAD-Q-L	Load Q register from left source
#x07	LOAD-Q-R	Load Q register from right source
#x08	NOT-L	One's complement left source
#x09	NOT-R	One's complement right source
#x0A	NEG-L	Two's complement left source
#x0B	NEG-R	Two's complement right source
#x0C	PRIORITIZE-L	Prioritize left source
#x0D	PRIORITIZE-R	Prioritize right source

#x0E	MERGE-L	Merge byte left into right
#x0F	MERGE-R	Merge byte right into left
#x10	L-1	Decrement left source by 1
#x11	R-1	Decrement right source by 1
#x12	L+1	Increment left source by 1
#x13	R+1	Increment right source by 1
#x14	L-2	Decrement left source by 2
#x15	R-2	Decrement right source by 2
#x16	L+2	Increment left source by 2
#x17	R+2	Increment right source by 2
#x18	L-4	Decrement left source by 4
#x19	R-4	Decrement right source by 4
#x1A	L+4	Increment left source by 4
#x1B	R+4	Increment right source by 4
#x1C	LOAD-STATUS-L	Load Status register from left source
#x1D	LOAD-STATUS-R	Load Status register from right source
#x1E		*Reserved*
#x1F		*Reserved*
#x20	SHIFT-DN-0F-L	Right-shift L one bit, inserting 0
#x21	SHIFT-DN-0F-R	Right-shift R one bit, inserting 0
#x22		Right-shift L:Q one bit, inserting 0
#x23		Right-shift R:Q one bit, inserting 0
#x24		Right-shift L one bit, inserting 1
#x25		Right-shift R one bit, inserting 1
#x26		Right-shift L:Q one bit, inserting 1
#x27		Right-shift R:Q one bit, inserting 1
#x28	SHIFT-DN-LF-L	Right-shift L one bit, inserting link
#x29	SHIFT-DN-LF-R	Right-shift R one bit, inserting link
#x2A		Right-shift L:Q one bit, inserting link
#x2B		Right-shift R:Q one bit, inserting link
#x2C	SHIFT-DN-AR-L	Right-shift L one bit, inserting sign
#x2D	SHIFT-DN-AR-R	Right-shift R one bit, inserting sign
#x2E		Right-shift L:Q one bit, inserting sign
#x2F	SHIFT-DN-AR-RQ	Right-shift R:Q one bit, inserting sign
#x30	SHIFT-UP-0F-L	Left-shift L one bit, inserting 0
#x31	SHIFT-UP-0F-R	Left-shift R one bit, inserting 0
#x32	SHIFT-UP-0F-LQ	Left-shift L:Q one bit, inserting 0
#x33	SHIFT-UP-0F-RQ	Left-shift R:Q one bit, inserting 0
#x34		Left-shift L one bit, inserting 1
#x35		Left-shift R one bit, inserting 1
#x36		Left-shift L:Q one bit, inserting 1
#x37		Left-shift R:Q one bit, inserting 1
#x38	SHIFT-UP-LF-L	Left-shift L one bit, inserting link
#x39	SHIFT-UP-LF-R	Left-shift R one bit, inserting link

#x3A		Left-shift L:Q one bit, inserting link
#x3B		Left-shift R:Q one bit, inserting link
#x3C	ZERO	Zero
#x3D	SIGN	Sign (-1 if n=1; 0 otherwise)
#x3E	OR	Logical OR
#x3F	XOR	Logical XOR
#x40	AND	Logical AND
#x41	XNOR	Logical Negated XOR
#x42	L+R *or* R+L	Add
#x43	L+R+C	Add with carry
#x44	L-R	Subtract right source from left source
#x45	L-R-C	Subtract right from left with carry
#x46	R-L	Subtract left source from right source
#x47	R-L-C	Subtract left from right with carry
#x48		BCD correct L for partial sum
#x49		BCD correct R for partial sum
#x4A		BCD correct L for partial difference
#x4B		BCD correct R for partial difference
#x4C		*Reserved*
#x4D		*Reserved*
#x4E	SDIV-FIRST	Signed divide, first step
#x4F		Unsigned divide, first step
#x50	SDIV-STEP	Signed divide, intermediate step
#x51	SDIV-LAST1	Signed divide, last step 1
#x52	MP-DIV-STEP1	Multiprecision divide, inner loop first step
#x53	MP-SDIV-STEP3	Signed multiprecision divide, inner loop last step
#x54		Unsigned divide, intermediate step
#x55		Unsigned divide, last step 1
#x56	MP-DIV-STEP2	Multiprecision divide, inner loop interm. step
#x57	MP-UDIV-STEP3	Unsigned multiprecision divide, inner loop last step
#x58	REM-CORR	Signed and unsigned remainder correction
#x59	QUO-CORR	Signed quotient correction
#x5A	SDIV-LAST2	Signed divide, last step 2
#x5B	UMUL-FIRST	Unsigned multiply, first step
#x5C	UMUL-STEP	Unsigned multiply, intermediate step
#x5D	UMUL-LAST	Unsigned multiply, last step
#x5E	SMUL-STEP	Signed multiply, intermediate step
#x5F	SMUL-FIRST	Signed multiply, first step
#x60	NB-SHIFT-AR-L	N bit shift left source with sign fill
#x61	NB-SHIFT-AR-R	N bit shift right source with sign fill
#x62	NB-SHIFT-0F-L	N bit shift left source with zero fill
#x63	NB-SHIFT-0F-R	N bit shift right source with zero fill
#x64	ROTATE-L	N bit rotate left source
#x65	ROTATE-R	N bit rotate right source
#x66	EXTRACT-BIT-LEFT	Extract bit from left source

#x67	EXTRACT-BIT-RIGHT	Extract bit from right source
#x68	SET-BIT-LEFT	Set bit in left source
#x69	SET-BIT-RIGHT	Set bit in right source
#x6A	RESET-BIT-LEFT	Reset bit in left source
#x6B	RESET-BIT-RIGHT	Reset bit in right source
#x6C	SET-BIT-STAT	Set bit in Status register
#x6D	RESET-BIT-STAT	Reset bit in Status register
#x6E	ALIGNED-FIELD-NOT-RIGHT	Invert field of right source
#x6F	ALIGNED-FIELD-PASS-RIGHT	Test a field of the right source
#x70	FIELD-NOT	Insert non-aligned not-left into right
#x71	ALIGNED-FIELD-NOT-LEFT	Insert aligned not-left into right
#x72	FIELD-PASS	Insert non-aligned left into right
#x73	ALIGNED-FIELD-PASS-LEFT	Insert aligned left into right
#x74	FIELD-OR	Logical OR of non-aligned field of left into right
#x75	ALIGNED-FIELD-IOR	Logical OR of aligned field of left into right
#x76	FIELD-XOR	Logical XOR of non-aligned field of left into right
#x77	ALIGNED-FIELD-XOR	Logical XOR of aligned field of left into right
#x78	FIELD-AND	Logical AND of non-aligned field of left into right
#x79	ALIGNED-FIELD-AND	Logical AND of aligned field of left into right
#x7A	FIELD-EXTRACT-L	Extract field from left source
#x7B	FIELD-EXTRACT-R	Extract field from right source
#x7C	FIELD-EXTRACT-LR	Extract field from L:R
#x7D	FIELD-EXTRACT-RL	Extract field from R:L
#x7E	EXTRACT-BIT-STATUS	Extract bit from Status register
#x7F		Pass mask

13.1 Notes and Caveats

1. Not all of the opcodes are implemented in the assembler. The relevant files are ORSON: FLEABIT.GENERATE; ASSEM LISP and K-SYS: K; ALU-OPCODES LISP.

2. There is something funny about the order of the signed multiply instructions.

3. There is no last-step-signed-multiply instruction. Consulting the AMD manual would probably be enlightening.

Part II

Memory Board Hardware

Chapter 14

Memory Board

This chapter is an overview of the memory board.

14.1 Overview

14.2 Memory Control Register

The Memory Control Register (MCR) is a 32-bit register. Its contents are used as control lines for various parts of the memory board. The MCR may be read or written by accessing functional source/destination 1100010.

All bits in the MCR are zeroed by a reset.

Bit(s)	*Meaning*
31	Master Trap Disable (0 = no trapping, 1 = trap under other masks). See below.
30	Asynchronous Trap Enable (0 = disable, 1 = enable)
29	Overflow Trap Enable (0 = disable, 1 = enable)
28	Data Type Trap Enable (0 = disable, 1 = enable)
27	Synchronous Trap Enable (0 = disable, 1 = enable)
26	Single step on trap exit (0 = disabled, 1 = enabled)
25	Spare
24	Reset Tap Bit (0 = reset trap bit on, 1 = normal)
23:20	Undefined
19	DRAM Parity Error Flagging (0 = disable, 1 = enable)

18	Boot PROM (0 = enabled, 1 = disabled)
17:16	Transporter RAM Mode Select
15	Use L or C valid/write-enable bits in map (0 = C bits, 1 = L bits)
14	Write Wrong Parity to DRAM (0 = normal parity, 1 = wrong parity)
13	16384 microsecond interrupt (0 = disable/reset request, 1 = enable)
12	1024 microsecond interrupt (0 = disable/reset request, 1 = enable)
11	I-Cache error clear (0 = disable/reset icache error traps, 1 = enable)
10:9	NuBus AD(1:0) bits for transfers
8	NuBus TM0 bit for transfers
7	LED 2 (0 = lit, 1 = unlit)
6	LED 1 (0 = lit, 1 = unlit)
5	LED 0 (0 = lit, 1 = unlit)
4	Statistics Source Polarity (0 = true, 1 = invert)
3:1	Statistics Counter Source (options listed below)
0	Statistics Counter Mode (0 = edge trigger, 1 = duration)

Note that bit 31 (Master Trap Disable) also sets/resets during trap exit/entry

Bits 3:1 determine the statistics counter source:

Value	*Source*
000	I-cache hit
001	Processor memory cycle
010	Instruction Status Bit
011	Undefined
100	PC in high core
101	Undefined
110	Undefined
111	Undefined

14.3 Memory Status Register

The Memory Status Register (MSR) is a 32-bit register. It contains various status bits from the memory board. The MSR may be read from functional source 1100110.

Bit(s)	*Meaning*
31:24	Undefined
23	Amount of memory installed (0 = 32 Meg, 1 = 16 Meg)
22	Autoboot jumper (0 = mastership external, 1 = go for it)
21	Memory Parity Error (0 = error, 1 = no error)
20:19	Undefined
18	MD Transport Trap (1 = MD read will cause transporter trap)

17	MD Page Trap (1 = MD read will cause read fault trap)
16	VMA Boxedness (0 = boxed, 1 = unboxed)
15	MD Boxedness (0 = boxed, 1 = unboxed)
14:13	Transporter Mode of Last Memory Cycle (see note)
12	Last Memory Cycle Type (0 = write, 1 = read)
11	Undefined
10:8	Nubus Bootstrap Mode (0 = normal, 1 = short reset, 2:7 = software)
7:4	ECO Jumper Number
3:0	Nubus Slot ID

Bits 14:13 – 00 = will write, 01 = no evcp, 10 = transport, 11 = no transport. See the Transporter RAM chapter for details.

14.4 Memory Board Hardware

The MCR [Memory Board, Page 22] is implemented with four 74LS273 registers. The four 74LS244 buffers are used for reading the MCR.

The MSR [Memory Board, Page 23] is implemented with four 74LS244 buffers.

Chapter 15

Main Memory Access

This chapter describes how the processor interacts with the virtual memory system.

15.1 Registers

The processor communicates with main memory via two 32-bit registers, the Virtual Memory Address (VMA) and the Memory Data (MD). When the processor wants to read or write data to main memory, it writes the appropriate address into the VMA, and it reads or writes the data in the MD.

Like the registers in register memory, both the VMA and the MD have box bits. However, these box bits do not behave like they do in register memory. When the processor writes a word from register memory into the VMA or MD, the box bit of that word is not loaded into the VMA/MD. Instead, the VMA/MD box bit is taken from bit 54 of the instruction. (See the Instruction Set chapter).

However, their box bits are registered separately registered separately. They are described in the Transporter RAM chapter.

15.2 Reading a Word from Memory

Reading a word from memory requires three instructions:

- Write the word's address to one of the eight VMA-START-READ functional destinations. The eight options allow the processor to choose one of four transporter modes (discussed below) and to optionally set the low bit of the address.
- Wait. This instruction can simply be a NOP, or it can do something else useful that doesn't read or write to the memory system.
- This instruction (and subsequent instructions, until the next memory system operation) can read the desired word from the MD functional source.

Before the processor sees the data in the MD, the Transporter RAM gets to look at it. Depending on which transporter mode was selected, the Transporter RAM will cause a trap if there is something wrong with the data in the MD. Here are the four transporter modes:

- **No Transport:** This mode should only be used when transporting unboxed data. No transporter trap will ever occur. (As a safety measure, if the MD is boxed, using no-transport mode will cause a trap).
- **Transport:** This is the normal way to transport boxed data. A transporter trap will occur if the MD includes a pointer into oldspace, if the MD is a forwarding pointer, if the MD has an invalid data type, or if the MD has data type $$DTP-UNBOUND.
- **Visible EVCP:** This mode is identical to Transport mode, except that forwarding pointers are not trapped. Instead, the data containing the forwarding pointer is left in the MD.
- **Will Write:** This mode is identical to Transport mode, except that unbounds are not trapped. Instead, the data with data type $$DTP-UNBOUND is left in the MD.

Note that the actual behavior of the Transporter RAM is not ingrained in the hardware. It depends on the contents of the RAM, which are downloaded at boot time. See the Transporter RAM chapter for more details.

The processor can force the low bit of the VMA to be 1 by using one of the VMA-START-READ-CDR functional destinations instead of the corresponding VMA-START-READ destination. When the processor knows that a pair of words are stored at an even address in virtual memory, and it knows the address of the first word, this option spares the processor the trouble of incrementing that address. This operation is useful when reading CONS and COMPLEX cells.

Here are the eight relevant functional destinations:

Value	*Abbreviation*
1110000	VMA-START-READ-NO-TRANSPORT
1110001	VMA-START-READ
1110010	VMA-START-READ-VISIBLE-EVCP
1110011	VMA-START-READ-WILL-WRITE
1110100	VMA-START-READ-CDR-NO-TRANSPORT
1110101	VMA-START-READ-CDR
1110110	VMA-START-READ-CDR-VISIBLE-EVCP
1110111	VMA-START-READ-CDR-WILL-WRITE

15.3 Writing a Word to Memory

To write a word to memory, the processor has to write an address into the VMA and the actual data into the MD. It can perform these operations in either order. For example, here is how the processor would write a word, writing the VMA first.

- Write the desired address to the VMA functional destination.
- Write the data to the MD-START-WRITE-NO-GC-TRAP or MD-START-WRITE functional destination.
- (& how long does the processor have to wait before starting another memory operation?)

The alternative method is this:

- Write the data to the MD functional destination.
- Write the desired address to the VMA-START-WRITE-NO-GC-TRAP or VMA-START-WRITE functional destination.

Here are the six relevant functional destinations:

Value	*Abbreviation*
1101100	VMA-START-WRITE-NO-GC-TRAP
1101101	VMA-START-WRITE
1101110	MD-START-WRITE-NO-GC-TRAP
1101111	MD-START-WRITE
110100X	VMA

110101X MD

There are two transporter modes for write operations, one which suppresses traps and one which enables them.

- **No GC Trap:** This transporter mode should be used for writing unboxed data.
- **Transport:** This transporter mode should be used for writing boxed data.

Again, note that the behavior of these modes depends on the contents of the Transporter RAM.

Chapter 16

Traps

The K processor has a single entry point for all traps and interrupts (location zero). Prioritizing of the trap causes is handled by software. When a trap occurs, any instruction that has passed the commit point will be completed. All others will be aborted and rerun after the trap return.

16.1 The Commit Point

Instructions have a logical point in their execution referred to as the commit point. Before this point the instruction can be aborted, any side effects undone, and then be re-run later.

Once the clock edge at the ALU/OREG boundary has occured, the commit point has been passed. The instruction then cannot be stopped from writing its destination.

16.2 Trap Entry

When a trap request occurs while traps are enabled, the processor clock that was about to occur will be delayed by one cycle. The CMEM1 clock edge that occurs wher the CPROC1 was about to occur will set the TRAP1 bit. This flag indicates the first state of a trap entry.

TRAP1 has multiple functions. It forces the PC to zero, the instruction cache is forced to restart its instruction access (possibly going to memory if the low-core cache is disabled), the call hardware will undo its previous function, the next two destination writes will be aborted, and the clock enables on some interesting registers will be turned off by the Trap State Machine (TSM).

16.3 Trap State Machine (TSM)

The TSM is a finite state machine that watches for certain PC values to occur, and then disables or enables the loading of certain hardware registers. The TSM wAtches for PCs in the range 0 to 31. This section of memory holds the trap entry and exit code.

The TSM also has the trace trap bit and several feedback bits as inputs to allow it to handle some special functions during trap exits.

16.4 Trap Entry Sequence

The following piece of assembly code is the instruction sequence that the TSM expects to find at location zero:

```
(defafun trap ()
;; The hardware depends on this loaded at location 0.
;; Save the oreg, source doesn't matter because pipeline
is shut off.
(alu setl gr::*save-oreg* r0 r0 bw-32 boxed-left)
;; Oreg clock comes on, we save the left alu input
(alu setl gr::*save-left* r0 r0 bw-32 boxed-left)
;; Left clock comes on, we save the right alu input
(alu setr gr::*save-right* r0 r0 bw-32 boxed-right)
;; Right clock comes on, we save the alu status
(alu pass-status gr::*save-status* r0 r0 bw-32 unboxed)
;; Alu clock comes on, we save the jump condition
(alu-field extract-bit-right gr::*save-jcond* r0
processor-status (byte 1.  (+ 32.  17.))  unboxed)
;; Jump condition clock comes on, find out which trap went
;; off.
(alu-field field-and gr::*save-trap* gr::*trap-mask*
trap-register (byte 31.  0.)  unboxed)
(alu prioritize-r gr::*trap-temp1* r0 gr::*save-trap*
bw-32 unboxed)
(alu-field set-bit-right gr::*trap-temp1* r0
gr::*trap-temp1* (byte 1.  5.)  unboxed)
;; Save pc
(alu merge-r gr::*save-trap-pc*  gr::*trap-dtp-code-5*
trap-pc bw-24 boxed)
;; Save pc + 1, dispatch to trap handler
(alu merge-r gr::*save-trap-pc+* gr::*trap-dtp-code-5*
trap-pc+  bw-24 boxed next-pc-dispatch)
```

The registers mentioned in the code will be re-enabled just after they are saved, thus allowing the processor to return to normal functionality as the sequence proceeds.

16.5 Normal Trap Exits (Non-modifying)

A normal trap exit will completely re-execute the trapped instruction. Since all of its side effects were undone when it was aborted, this doesn't cause any problem. The following instruction sequence is the normal trap exit code:

```
(defafun non-modifying-exit ()
;; The hardware depends on this loading at location 12.
;; Jump condition gets fed to magic flipflop.
(alu-field field-pass processor-control gr::*save-jcond*
processor-control (byte 1.  4.))
;; Restore status of trapped instruction, alu clock turns
;; off.
(alu load-status-r nop r0 gr::*save-status* bw-32)
;; Pipeline saved pc for returning.
(alu setl gr:*trap-temp1* gr::*save-trap-pc*
gr::*save-right*  bw-32 boxed-left)
;; Pipeline saved pc+ for restarting dispatch
;; instructions.
(alu setl gr:*trap-temp1* gr::*save-trap-pc+*
gr::*save-right*  bw-32 boxed-left)
;; Jump to trapped instruction, pipeline jump condition
;; for trapped jumps.
(alu setl gr:*trap-temp1* gr::*save-oreg* gr::*save-right*
bw-32 next-pc-dispatch br-jindir boxed-left)
```

16.6 Modifying Trap Exits

A modifying trap exit is a special exit used for datatype and overflow traps. If the datatype trap is serviceable, then the effective result of the aborted instruction will be computed and put into GR:*SAVE-RIGHT*, The status will be put into GR:*SAVE-STATUS*, and then the modifying exit code will be run. This will have the effect of mostly re-executing the trapped instruction. However, instead of the ALU result being written to the destination, the value in the right register will be written instead (the ALU opcode will be forced to PASS-RIGHT and the box code to BOXED-RIGHT). In addition, datatype traps will be suppressed during this instruction to prevent recausing the same trap.

This all has the effect of allowing the trap routine to substitute a result for the instruction. This is useful, for example, when adding two complex numbers.

The folowing is the non-modifying exit code sequence:

```
(defafun modifying-exit ()
;; The hardware depends on this loading at location 20.
;; Jump condition gets fed to magic flipflop.
(alu-field field-pass processor-control gr::*save-jcond*
processor-control (byte 1.  4.))
;; Restore status of trapped instruction, alu clock turns
;; off.
(alu load-status-r nop r0 gr::*save-status* bw-32)
;; Pipeline saved pc for returning.  Right side clock shuts
;; off, save right gets caught.
(alu setl gr:*trap-temp1* gr::*save-trap-pc*
gr::*save-right* bw-32 boxed-left)
;; Pipeline saved pc+1 for dispatches.
(alu setl gr:*trap-temp1* gr::*save-trap-pc+*
gr::*save-right* bw-32 boxed-left)
;; Jump to trapped instruction, setup saved jump
;; condition.
(alu setl gr:*trap-temp1* gr::*save-oreg* gr::*save-right*
bw-32 next-pc-dispatch br-jindir boxed-left)
```

16.7 Diagnostic Trap Exits

This code sequence is used only for instruction cache diagnostics. It allows the data in the cache to be read and held in the cache transceiver registers. It can be read later for running cache diagnostics. This can also be used to force cache locations to be accessed and loaded from memory (as in initializing the low-core cache). The exit sequence proceeds only far enough to allow the cache to fetch the desired instruction. Then the single step trap will return control to the trap handler.

```
(defafun diagnostic-trap-exit ()
;; The hardware depends on this assembling at location 28.
;; It causes a trap to happen after the instruction fetch
;; but before the instruction register gets loaded.    This
;; enables us to run icache diagnostics.
;; Dispatch to trap pc.
(alu setl nop gr::*save-trap-pc*  gr::*save-right* bw-32)
;; This instruction can be ignored.
(alu setl nop gr::*save-trap-pc+* gr::*save-right* bw-32)
;; This just does a dispatch.
(move nop gr::*save-oreg* bw-32 next-pc-dispatch)
```

16.8 Trace Trapping

The TSM handles the trace trap function. If the trace trap bit in the control register is set when a trap exit sequence is executed, then the TSM will cause a trap requested just after the trapped instruction has passed its commit point.

Note that some other trap can still come in before the commit point. However, when that trap handler returns the trace trap will occur.

The trace trap will occur early in the case of the diagnostic exit since we want to abort very early in the instruction.

16.9 Trap Vector Table

This is the 32-entry vector table that the trap entry code will branch to after saving its registers. They correspond to the bits of the trap register.

```
(defafun trap-vector-table () ;;; at absolute location 32.
;; This "function" is actually a dispatch table.
trap-vector-reset       ;Bit 31 - addr 32 - Highest priority
(jump reset-trap-handler ())
trap-vector-trace       ;Bit 30 - addr 33
(jump trace-trap-handler ())
trap-vector-icache-parity    ;Bit 29 - addr 34
(jump icache-parity-trap-handler ())
trap-vector-icache-nubus-err    ;Bit 28 - addr 35
(jump icache-nubus-error-trap-handler ())
trap-vector-icache-nubus-timeout    ;Bit 27 - addr 36
(jump icache-nubus-timeout-trap-handler ())
trap-vector-icache-page-fault    ;Bit 26 - addr 37
(jump icache-map-fault-trap-handler ())
trap-vector-proc-mread-parity    ;Bit 25 - addr 38
(jump memory-read-parity-trap-handler ())
trap-vector-proc-mread-nubus-err    ;Bit 24 - addr 39
(jump memory-read-nubus-error-trap-handler ())
trap-vector-proc-mread-nubus-timeout   ;Bit 23- addr 40
(jump memory-read-nubus-timeout-trap-handler ())
trap-vector-proc-mread-page-fault   ;Bit 22 - addr 41
(jump memory-read-page-fault-trap-handler ())
trap-vector-proc-mread-transporter   ;Bit 21 - addr 42
(jump memory-read-transporter-trap-handler ())
trap-vector-proc-mwrite-nubus-err   ;Bit 20 - addr 43
(jump memory-write-nubus-error-trap-handler ())
trap-vector-proc-mwrite-nubus-timeout   ;Bit 19-  addr 44
(jump memory-write-nubus-timeout-trap-handler ())
trap-vector-proc-mwrite-page-fault   ;Bit 18 - addr 45
(jump memory-write-page-fault-trap-handler ())
trap-vector-proc-mwrite-gc    ;Bit 17 - addr 46
(jump memory-write-gc-trap-handler ())
trap-vector-floating-point    ;Bit 16 - addr 47
(jump floating-point-trap-handler ())
trap-vector-heap-empty     ;Bit 15 - addr 48
(jump heap-empty-trap-handler ())
trap-vector-instruction-bit    ;Bit 14 - addr 49
(jump instruction-trap-handler ())
trap-vector-datatype     ;Bit 13 - addr 50
(jump datatype-trap-handler ())
trap-vector-overflow     ;Bit 12 - addr 51
(jump overflow-trap-handler ())
trap-vector-spare11     ;Bit 11 - addr 52
(jump spare11-trap-handler ())
trap-vector-interrupt7     ;Bit 10 - addr 53
(jump debugger-trap-handler ())
trap-vector-interrupt6     ;Bit 09 - addr 54
(jump interrupt6-trap-handler ())
trap-vector-interrupt5     ;Bit 08 - addr 55
(jump interrupt5-trap-handler ())
trap-vector-interrupt4     ;Bit 07 - addr 56
(jump iop-trap-handler ())
trap-vector-interrupt3     ;Bit 06 - addr 57
(jump interrupt3-trap-handler ())
trap-vector-interrupt2     ;Bit 05 - addr 58
(jump interrupt2-trap-handler ())
trap-vector-interrupt1     ;Bit 04 - addr 59
(jump interrupt1-trap-handler ())
trap-vector-interrupt0     ;Bit 03 - addr 60
(jump interrupt0-trap-handler ())
trap-vector-timer-1024     ;Bit 02 - addr 61
(jump timer-1024-trap-handler ())
trap-vector-timer-16384     ;Bit 01 - addr 62
(jump timer-16384-trap-handler ())
trap-vector-spurious     ;Bit 00 - addr 63
(jump spurious-trap-handler ()))
```

Chapter 17

Transporter RAM

This chapter describes the purpose and function of the transporter RAM.

17.1 Introduction

The Transporter RAM is a 4K x 4 RAM. Its purpose is to examine the data type of anything that appears on the Memory Data (MD) bus, and cause a trap if something is amiss. This decision is also based on the type of memory cycle which caused the data to appear in the MD, and two control bits in the Memory Control Register.

17.2 Input Lines

The contents of the Transporter RAM are set up at boot time. The contents should probably not be altered unless the processor is being rebooted. The input lines are connected to bits 7:4 of the MMFIO bus; the RAM is loaded by cleverly setting up the address lines, then writing to functional destination 1100101. It requires a small amount of wizardry to set up the address lines at boot time; see the function K-ADDRESS-TRANSPORTER-RAM in the JB: KBUG; NEW-SPY-UTILITIES.LISP file for an example.

17.3 Address Lines

There are twelve address lines. The top four are decoded from the boxedness and functional destination fields of the relevant instruction. The middle two are obtained from the Memory Control Register. The bottom six come from the MD bus itself.

17.3.1 Boxedness (Bits 11 and 10)

Bit 11 is the negation of bit 54 of the most recent instruction which wrote to any VMA-START-READ, VMA-START-WRITE, or VMA functional destination.

Bit 10 is the negation of bit 55 of the most recent instruction which wrote to any VMA-START-READ, MD-START-WRITE, or MD functional destination.

Both of these bits are registered [Memory board, PREQ PAL]. They are updated each time an appropriate functional destination is written.

17.3.2 Memory Cycle Type (Bits 9 and 8)

These two address lines depend on the transporter mode requested by the most recent memory instruction. The last memory cycle type is registered [Memory board, PREQ PAL]. It is updated every time a VMA or MD functional destination is written.

Last Memory Cycle Type	*Bit 9 (PMT1)*	*Bit 8 (PMT0)*
Read, no Transport	0	0
Read with Transport	0	1
Read, visible EVCP	1	0
Read, will write	1	1
Write (VMA), no GC trap	0	0
Write (VMA)	0	1
Write (MD), no GC trap	1	0
Write (MD)	1	1

These bits are represented with negative logic; they therefore appear negated on the address lines of the Transporter RAM. These two (negative logic) bits are also loaded into bits 14:13 of the MSR.

It is worth mentioning that a third bit is also registered by the PREQ PAL; this bit is 0 if the last memory cycle type was a read operation and 1 if the last memory cycle type was a write operation. This bit is also represented with negative logic. (It is called PROCWRIT on the design sheets.) It is loaded (also negated) into bit 12 of the MSR. The Transporter RAM ignores this bit. However, it influences some decisions of the trap logic (described below).

17.3.3 MCR Bits (Bits 7 and 6)

Bits 7:6 of the Memory Control Register. Right now, the only value with any meaning is 00. The other values may be needed when implementing the garbage collector.

17.3.4 Data Type (Bits 5 to 0)

Bits 31:26 (the data type field) of the MD.

17.4 Output Lines

The Transporter RAM outputs four useful signals. Two of them (Trappable Pointer and Box Error) have effect during memory writes. The other two have effect during memory reads.

17.4.1 Trappable Pointer

During a memory write, if the trappable pointer line is asserted, a volatility comparison takes place.

17.4.2 Trap if Old

Indicates that a trap should occur if bits 25:0 of the MD point into oldspace.

17.4.3 Trap if New

Indicates that a trap should occur if bits 25:0 of the MD point into newspace.

17.4.4 Box Error

Forces a GC (Garbage Collector) trap.

17.5 Trap Logic

17.6 Contents of the Transporter RAM

For each combination of VMA box bit, MD box bit, data type, and control mode, there are sixteen bits in the Transporter RAM. The sixteen bits are a product of four transporter modes and four output lines. The modes, numbered from 00 to 11, have different meanings for read and write operations. Two of the output lines are used only during read operations, and two are used only during write operations.

17.6.1 Read Operations

Eight of the bits affect read operations:

MODE	OUTPUTS			
00 No Transport	Ignored	Trap If New	Trap If Old	Ignored
01 Transport	Ignored	Trap If New	Trap If Old	Ignored
10 Visible EVCP	Ignored	Trap If New	Trap If Old	Ignored
11 Will Write	Ignored	Trap If New	Trap If Old	Ignored

17.6.2 Write Operations

Eight of the bits affect write operations. Note that there are two encodings for each transporter mode. One encoding is used when the VMA is written before the MD during a write operation; the other encoding is used when the MD is written first. Since the order of writing the MD and VMA should not affect the transporter's behavior, it is very important that these outputs be the same for both encodings.

```
MODE                                   OUTPUTS
                       +---------+---------+---------+---------+
00  No GC Trap         |Box Error| Ignored | Ignored |Trappable|
                       |         |         |         | Pointer |
                       +---------+---------+---------+---------+
01  Transport          |Box Error| Ignored | Ignored |Trappable|
                       |         |         |         | Pointer |
                       +---------+---------+---------+---------+
10  No GC Trap         |Box Error| Ignored | Ignored |Trappable|
                       |         |         |         | Pointer |
                       +---------+---------+---------+---------+
11  Transport          |Box Error| Ignored | Ignored |Trappable|
                       |         |         |         | Pointer |
                       +---------+---------+---------+---------+
```

17.6.3 Patterns

It is always an error to read boxed data using the no-transport mode or write boxed data using the no-gc-trap mode. Therefore, every transporter entry for boxed data should cause a trap in these two situations. The first chart shows the bit patterns which trap on illegal reads; the second chart shows the bit patterns which trap on illegal writes.

```
          OUTPUTS                  OUTPUTS
   +---+---+---+---+        +---+---+---+---+
00 |   | 1 | 1 |   |     00 | 1 |   |   |   |
   +---+---+---+---+        +---+---+---+---+
01 |   |   |   |   |     01 |   |   |   |   |
   +---+---+---+---+        +---+---+---+---+
10 |   |   |   |   |     10 | 1 |   |   |   |
   +---+---+---+---+        +---+---+---+---+
11 |   |   |   |   |     11 |   |   |   |   |
   +---+---+---+---+        +---+---+---+---+
```

17.7 Setting up the Transporter RAM

The Transporter RAM setup routines are in the file JB: K; TRANSPORTER-RAM.LISP. The function LOAD-TRANSPORTER-RAM-DATA actually sets up the Transporter RAM pattern. It makes repeated calls to the function LOAD-TRANSPORTER-RAM-PATTERN, which takes six arguments: (1) The VMA box bit, (2) The MD box bit, (3) The data type, (4) The transporter mode, (5) The control mode from the MCR, (6) The value to load into the Transporter RAM. Any of the first five arguments may be `t`, which acts as a wildcard.

The sixth argument is a four-digit binary number. From MSB to LSB, the bits represent the values to place on the: (3) Box Error, (2) Trap if New, (1) Trap if Old, and (0) Trappable Pointer output lines.

The default is to trap on anything unusual:

```
(load-transporter-ram-pattern t t t                t        t      #b1111)
```

The following cases handle unboxed data. Unboxed data is indicated by the VMA and MD box bits both being zero. Reading unboxed data should not cause a trap in no-transport mode. Writing unboxed data should not cause a trap in the two no-gc-trap modes.

```
(load-transporter-ram-pattern 0 0 t                no-trans normal #b0000)
(load-transporter-ram-pattern 0 0 t                vis-evcp normal #b0110)
```

The following pattern is used for Lisp values which do not contain pointers. These include NIL, fixnums, and characters.

```
(load-transporter-ram-pattern t 1 vinc:$$dtp-nil    no-trans normal #b1111)
(load-transporter-ram-pattern t 1 vinc:$$dtp-nil    trans    normal #b0000)
(load-transporter-ram-pattern t 1 vinc:$$dtp-nil    vis-evcp normal #b1001)
(load-transporter-ram-pattern t 1 vinc:$$dtp-nil    write    normal #b0000)
```

The following pattern is used for Lisp values which contain pointers. These include arrays, structures, hash tables, cons cells, symbols, bignums, rationals, short floats, single floats, double floats, and complex numbers.

```
(load-transporter-ram-pattern t 1 vinc:$$dtp-bignum   no-trans normal #b1111)
(load-transporter-ram-pattern t 1 vinc:$$dtp-bignum   trans    normal #b0011)
(load-transporter-ram-pattern t 1 vinc:$$dtp-bignum   vis-evcp normal #b1011)
(load-transporter-ram-pattern t 1 vinc:$$dtp-bignum   write    normal #b0011)
```

Chapter 18

Garbage Collector (GC) RAM

GC RAM ADDRESS IS MD(25:14)

Chapter 19

Spy Hardware

Part III

Lisp Software

Chapter 20

Storage Conventions

This chapter defines the storage conventions used by the K processor for Lisp objects.

20.1 Structure of Data Words

Words in the machine consist of 33 bits. The most significant bit (bit 32) is referred to as the BOX bit. When zero, the other 32 bits of the word contain an untyped 32 bit number. The only software which should explicitly manipulate unboxed data is internal system routines.

32	31 … 0
0	Unboxed Data

When the box bit is set, the word is a typed LISP value. The most significant 6 bits (31:26) indicate the data type of the word. Bits 25:0 vary in meaning with the data type. Very often, bits 25:0 are used as a pointer into virtual memory. Hence, this field is often called the ”pointer” field.

32	31 … 26	25 … 0
1	Data Type	Pointer

20.2 Tables of Data Types

The six DTP (Data Type) bits can represent up to 64 distinct types. The data type definitions currently reside in K-SYS: K; DATA-TYPES LISP. By convention, data types 0 to 31 are visible data types. (They identify valid Lisp objects.) Data types 32 to 63 are invisible data types used by the internal system routines. This distinction is arbitrary and is not enforced by the hardware.

Data types 26 through 31 and 49 through 63 are currently unassigned.

20.2.1 Visible Data Types

DTP (Decimal)	*DTP (Binary)*	*Data Type*
0	000000	$$DTP-NIL
1	000001	$$DTP-FIXNUM
2	000010	$$DTP-CONS
3	000011	$$DTP-SYMBOL
4	000100	$$DTP-BIGNUM
5	000101	$$DTP-SHORT-FLOAT
6	000110	$$DTP-SINGLE-FLOAT
7	000111	$$DTP-DOUBLE-FLOAT
8	001000	$$DTP-RATIONAL
9	001001	$$DTP-COMPLEX
10	001010	$$DTP-LOCATIVE
11	001011	$$DTP-UNBOXED-LOCATIVE
12	001100	$$DTP-COMPILED-FUNCTION
13	001101	$$DTP-CODE
14	001110	$$DTP-ARRAY
15	001111	$$DTP-STACK-GROUP
16	010000	$$DTP-INSTANCE
17	010001	$$DTP-LEXICAL-CLOSURE
18	010010	$$DTP-INTERPRETER-CLOSURE
19	010011	$$DTP-LEXICAL-ENVIRONMENT
20	010100	$$DTP-STRUCTURE
21	010101	$$DTP-CHARACTER
22	010110	$$DTP-EXTEND
23	010111	$$DTP-ENCAPSULATION
24	011000	$$DTP-HASH-TABLE

20.2.2 Invisible Data Types

DTP *(Decimal)*	*DTP* *(Binary)*	*Data Type*
32	100000	$$DTP-UNBOXED-HEADER
33	100001	$$DTP-SYMBOL-HEADER
34	100010	$$DTP-ARRAY-HEADER-SINGLE
35	100011	$$DTP-ARRAY-HEADER-MULTIPLE
36	100100	$$DTP-ARRAY-HEADER-EXTENSION
37	100101	$$DTP-EXTERNAL-VALUE-CELL-POINTER
38	100110	$$DTP-GC-FORWARD
39	100111	$$DTP-ONE-Q-FORWARD
40	101000	$$DTP-INDEXED-FORWARD
41	101001	$$DTP-INSTANCE-HEADER
42	101010	$$DTP-ARRAY-LEADER-HEADER
43	101011	$$DTP-UNBOUND
44	101100	$$DTP-HEADER-FORWARD
45	101101	$$DTP-BODY-FORWARD
46	101110	$$DTP-COMPILED-FUNCTION-HEADER
47	101111	$$DTP-STRUCTURE-HEADER
48	110000	$$DTP-HASH-TABLE-HEADER

20.3 Numbers

20.3.1 Fixnums

Fixnums have a data type of $$DTP-FIXNUM. Bits 23:0 of the pointer field contain a 24-bit two's complement value. Bits 25:24 are unused and should be zero.

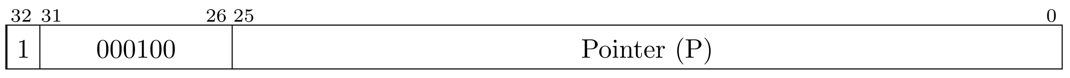
Fixnums have a range of -2^{23} through $2^{23} - 1$.

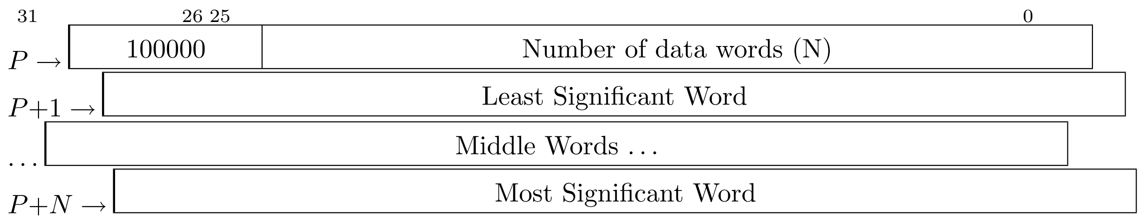
32	31 26	25 24	23 0
1	000001	00	24-Bit Two's Complement Value

20.3.2 Bignums

Bignums have a data type of $$DTP-BIGNUM. A bignum is a pointer to a memory structure. The memory structure consists of a header word with datatype $$DTP-UNBOXED-HEADER and a pointer field containing the number of words of storage used for the bignum data words. The words after the header form an N-word two's complement number stored least significant word first. Bignum storage is allocated in structure space.

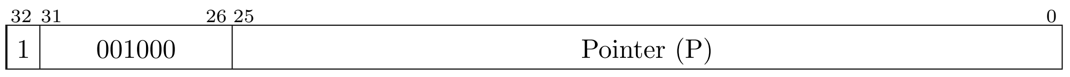
Bignums may use up to 2^{18} words of storage for a number. The range of bignums is $-2^{2^{23}}$ to $2^{2^{23}} - 1$. However, a bignum may not have a value in the fixnum range (-2^{23} to $2^{23} - 1$). If the result of a bignum operation lies in that range, it must be converted to a fixnum.

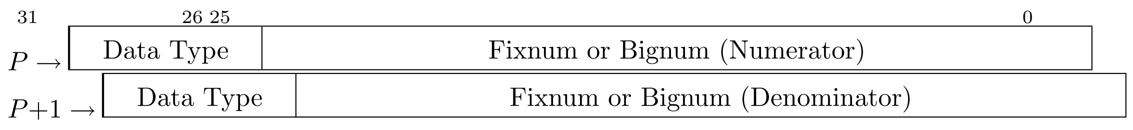




20.3.3 Rationals

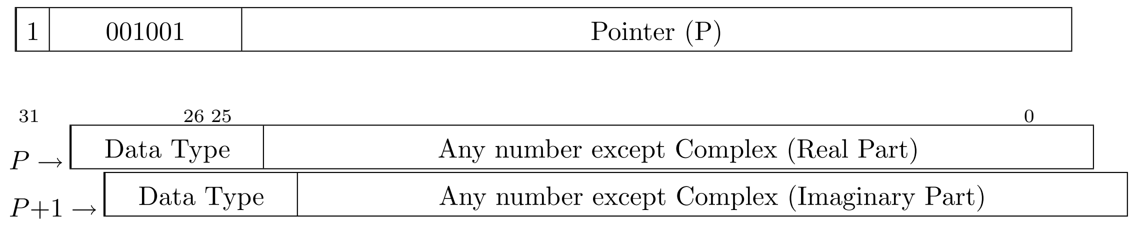
Rational numbers consist of a numerator and a denominator. Each may be either a fixnum or a bignum. A rational has a data-type of $$DTP-RATIONAL. The pointer field points to a pair of words in memory which contain the numerator and denominator. The storage for rationals is allocated in CONS space.





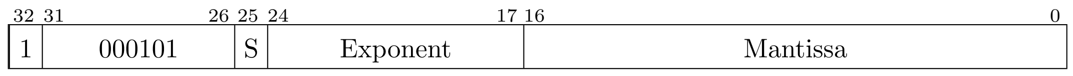
20.3.4 Complex

Complex numbers consist of a real and an imaginary part. Each may be of any numeric type other than Complex. A complex has a data type of $$DTP-COMPLEX. The pointer field points to a pair of words in memory, the real and imaginary parts of the number. The storage for complex numbers is allocated in CONS space.


1
001001
Pointer (P)
31
26 25
0
P →
Data Type
Any number except Complex (Real Part)
P+1 →
Data Type
Any number except Complex (Imaginary Part)

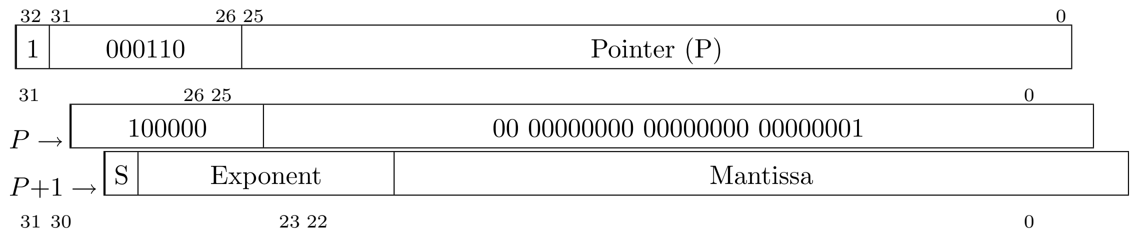
20.3.5 Short Floating Point

Short floating point numbers have a data type of $$DTP-SHORT-FLOAT. The pointer field contains the high order 26 bits of an IEEE single precision floating point number. This includes a sign bit, an 8-bit excess 127 exponent, and a 17-bit mantissa.



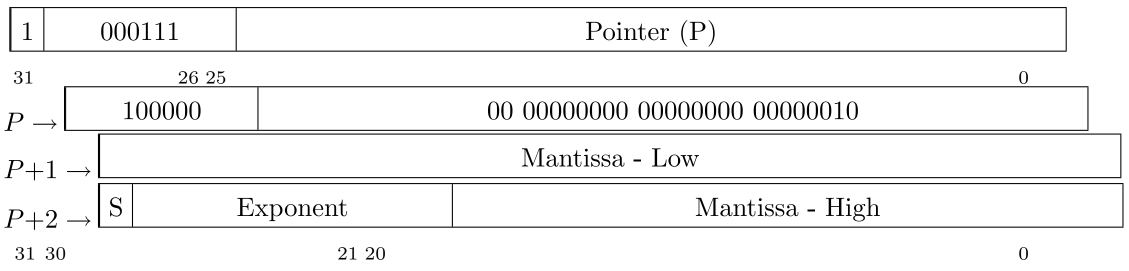
20.3.6 Single Precision Floating Point

A single float value has a data-type of $$DTP-SINGLE-FLOAT. The pointer field points to a memory structure introduced by an unboxed header ($$DTP-UNBOXED-HEADER) whose pointer field contains a length of one. The data word contains a single precision IEEE floating point number.



20.3.7 Double Precision Floating Point

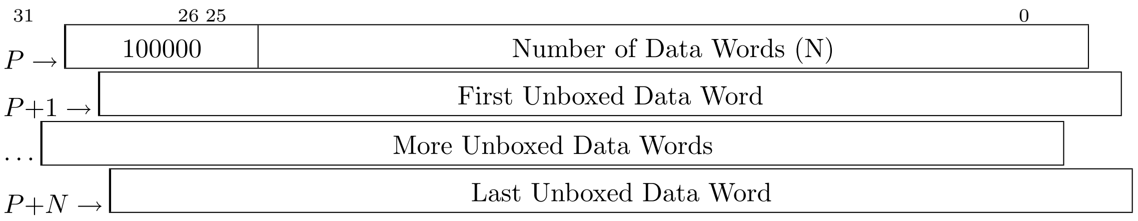
A double float value has a data type of $$DTP-DOUBLE-FLOAT. The pointer field points to a memory structure introduced by an unboxed header ($$DTP-UNBOXED-HEADER) whose pointer field contains a length of two. The first data word contains the least significant part of an double precision IEEE floating point number. The second word contains the most significant part of the number.



20.4 Unboxed Structures

Unboxed structures are used for storage of untyped non-array data in memory. Unboxed structures do not indicate what kind of data they are storing; this is determined by the data type of the Lisp object which points to the unboxed structure. For exampes of data types which use unboxed structures, see bignums, single floats, and double floats.

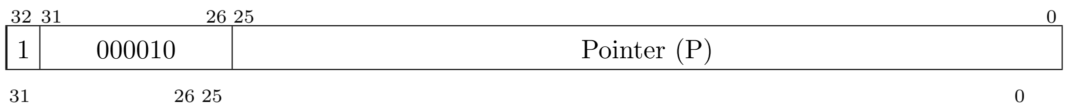
An unboxed structure is introduced by a word of type $$DTP-UNBOXED-HEADER whose pointer field contains the number of data words following the header. In the following chart, P is the pointer field of the Lisp object which points to the unboxed structure.

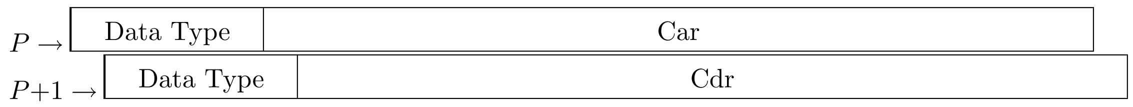


20.5 Characters

20.6 Conses

A cons has a data type of $$DTP-CONS. Its pointer field points to a pair of words in memory, the car and cdr of the cons cell. The storage for conses is allocated in CONS space.

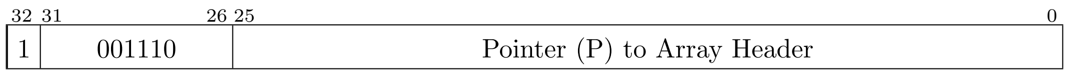




20.7 Arrays

Arrays have two related storage formats, simple arrays and hard arrays. Simple arrays are one-dimensional arrays that are not adjustable, not displaced, and do not have fill pointers or leaders. Hard arrays (also referred to as "full" or "multiple" arrays) can have up to seven dimensions, can be adjustable, can be displaced, and can have leaders and fill pointers. Simple arrays are faster to access and require less overhead for storage than do hard arrays.

All arrays consist of an array pointer of type $$DTP-ARRAY which points to the header of the array. The header either has type $$DTP-ARRAY-HEADER-SINGLE, for simple arrays, or $$DTP-ARRAY-HEADER-MULTIPLE, for hard arrays. The array's data begins in the word following the header. If the array is hard, then there are more words of header information preceding the header word in memory.



20.7.1 Array Element Types

The array header information contains a five-bit field called the "Array Type". This number describes the types of elements the array can contain. In simple arrays, this number is contained in bits 25:21 of the array header. In hard arrays, bits 25:21 of the array header are 11111, and the array type is found in bits 13:9 of the extension header.

Type	*Bits per Element*	*Array Type*	*Range or Type of Elements*
ART-Q	32	00000	Any Lisp object
ART-1B	1	00001	0 to 1
ART-2BS	2	00010	-2 to 1
ART-2B	2	00011	0 to 3
ART-4BS	4	00100	-8 to 7
ART-4B	4	00101	0 to 15
ART-8BS	8	00110	-128 to 127
ART-8B	8	00111	0 to 255

ART-16BS	16	01000	-32768 to 32767
ART-16B	16	01001	0 to 65536
ART-32BS	32	01010	-2147483648 to 2147483647
ART-32B	32	01011	0 to 4294967296
ART-STRING	8	01100	Characters
ART-FAT-STRING	16	01101	Characters w/fonts
ART-SINGLE-FLOAT	32	01110	Single precision floats
ART-DOUBLE-FLOAT	64	01111	Double precision floats
ART-CONTROL-PDL	??	11100	??
ART-EXTRANEOUS-PDL	??	11101	??
ART-SPECIAL-PDL	??	11110	??
ART-HARD	??	11111	In extension header

ART-Q

An ART-Q (array type 00000) array can store any Lisp object in each array location. The elements of the array use one word of storage each.

The array type of ART-Q is 00000 as an optimization for array reference software since this is the most common array type.

Bit array types

Bit arrays store either a signed or unsigned integer in each element. Types that are smaller than a word have multiple elements packed into each word. The element with the lowest array index is stored in the least significant bits of the word. When a value is read by the @l[AREF] function, it is converted to a fixnum (except for large 32 bit numbers, which are converted to bignums). The signed bit array types are stored as two's complement numbers.

Bit arrays have array types from 00001 to 01001.

Strings

There are two array types used for strings, ART-STRING (array type 01100) and ART-FAT-STRING (array type 01101). Elements of ART-STRING arrays are eight-bit characters. Elements of ART-FAT-STRING arrays are sixteen-bit characters (they also allow font and bit information).

ART-STRING arrays are stored like ART-8B arrays. However, data read out is converted to a character instead of a fixnum. This array type is the Common Lisp string type.

ART-FAT-STRING arrays are stored like ART-16B arrays. Data read out is also converted to characters, but the high 8 bits are now converted into font and bit information for the character.

Floating point

There are two types of floating point arrays, ART-SINGLE-FLOAT (array type 01110) and ART-DOUBLE-FLOAT (array type 01111). Both hold IEEE standard format floating point values. The ART-SINGLE-FLOAT arrays use one word to store a number, whereas ART-DOUBLE-FLOAT arrays use two words to contain a value. The least significant part of the double precision number is stored in the first word, and the most significant part is stored in the second word.

20.7.2 Format of Array Data

Depending on the array type, an array element can be anywhere from 1 bit to 64 bits long. In general, if there are N elements in an array, and each element requires B bits of storage, the number of words in the array's data will be $S = \lceil NB/32 \rceil$, where $\lceil x \rceil$ represents the lowest integer greater than or equal to x.

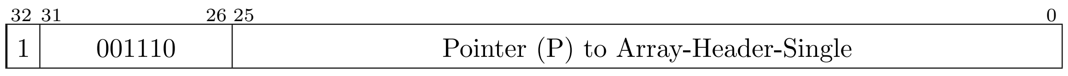
The elements are laid out in memory with lower indices corresponding to lower memory locations. Within a word, lower array indices correspond to less significant bits. Multidimensional arrays are arranged in row-major form. For a detailed description of how these arrays are laid out, see *Common Lisp, the Language*, pp. 286-298.

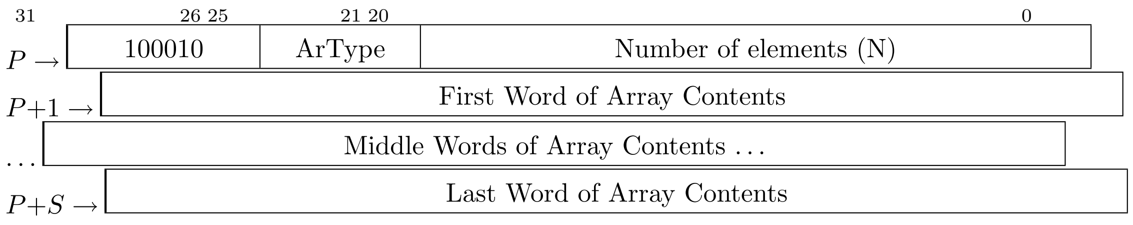
(& example?)

20.7.3 Simple Arrays

Simple arrays are one dimensional arrays (”vectors”) of any of the allowed array types. They are not adjustable, not displaced, and do not have fill pointers or leaders. A simple array has data type $$DTP-ARRAY (001110). Its pointer field points to a header with data type $$DTP-ARRAY-HEADER-SINGLE

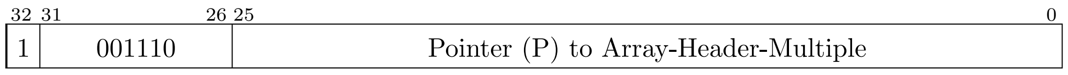
(100010). Bits 25:21 of the header contain the array type. Bits 20:0 of the header indicate the number of elements in the array.

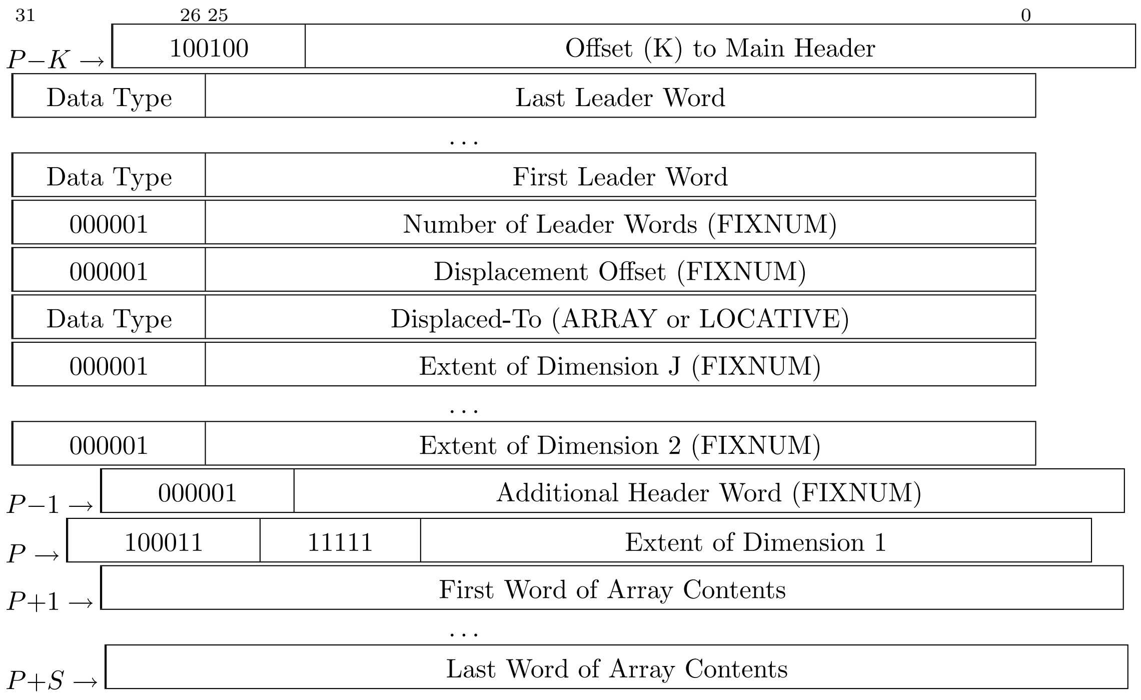



31
26 25
21 20
0
P →
100010
ArType
Number of elements (N)
P+1 →
First Word of Array Contents
...
Middle Words of Array Contents ...
P+S →
Last Word of Array Contents

20.7.4 Hard Arrays

Hard arrays have a more complicated format than do simple arrays. A hard array has data type $$DTP-ARRAY (001110). Its pointer field points to a header with data type $$DTP-ARRAY-HEADER-MULTIPLE (100011). The array's data follows the header. Additional array information appears in memory locations *before* the header. This additional information is introduced with a header of data type $$DTP-ARRAY-HEADER-EXTENSION (100100), and it extends from the array-header-extension header up to the array-header-multiple header.




31
26 25
0
P−K →
100100
Offset (K) to Main Header
Data Type
Last Leader Word
...
Data Type
First Leader Word
000001
Number of Leader Words (FIXNUM)
000001
Displacement Offset (FIXNUM)
Data Type
Displaced-To (ARRAY or LOCATIVE)
000001
Extent of Dimension J (FIXNUM)
...
000001
Extent of Dimension 2 (FIXNUM)
P−1 →
000001
Additional Header Word (FIXNUM)
P →
100011
11111
Extent of Dimension 1
P+1 →
First Word of Array Contents
...
P+S →
Last Word of Array Contents

Extension Header Format

An array header extension has data type $$DTP-FIXNUM (000001). It has the following format:

Bit(s)	*Meaning*
31:26	Data Type field (000001)
25:23	Spare
22	Is the array adjustable? (0 = no, 1 = yes)
21	Does the array have a fill pointer? (0 = no, 1 = yes)
20	Is the array a named structure? (0 = no, 1 = yes)
19	Is the array displaced? (0 = no, 1 = yes)
18	Spare
17	Does the array have a leader? (0 = no, 1 = yes)
16:14	Number of array dimensions (J)
13:9	Array type. 11111 is an error.
8:4	Spare
3:0	Leader offset (L)

Fill Pointer

The fill pointer is a fixnum that indicates the number of words currently used in an array. If present, it is part of the leader.

Displaced Arrays

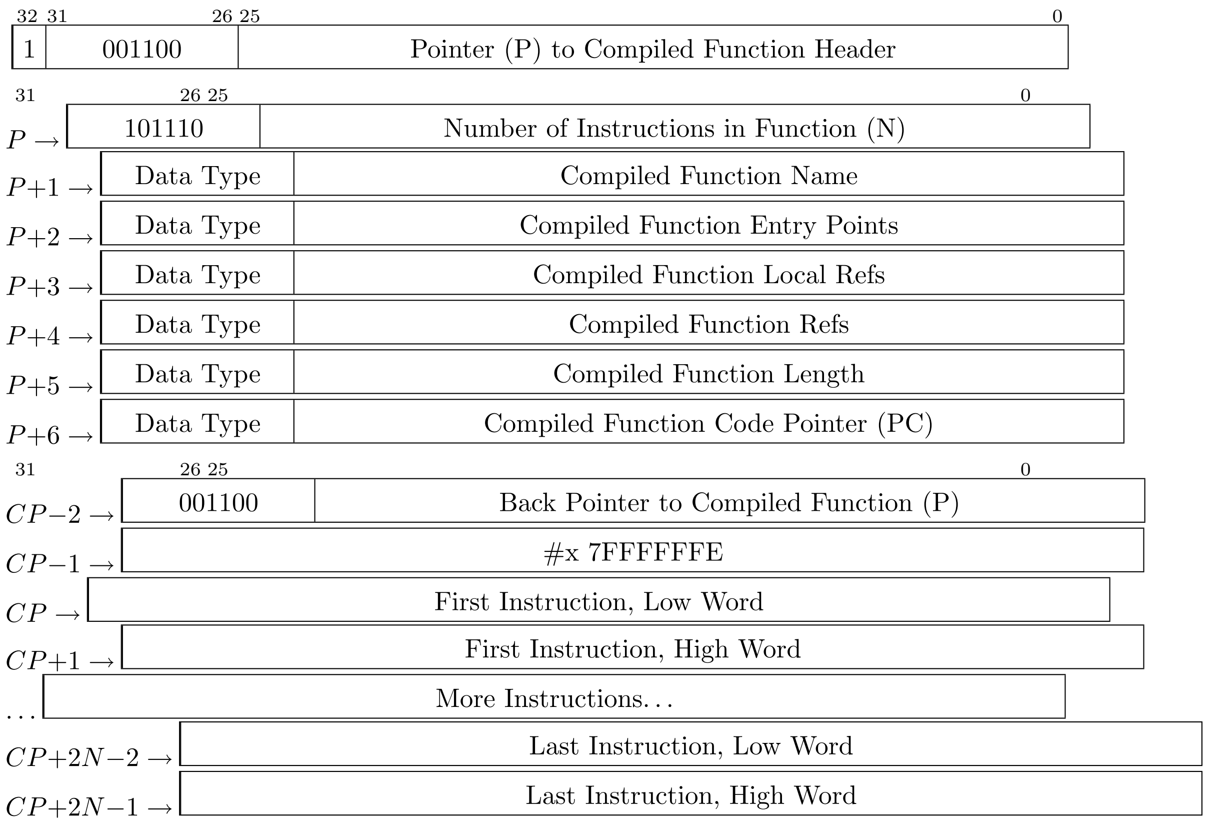
Leaders

If the has leader bit is set then the array has an array leader. The offset from the main header to the first word of the leader is in the leader-offset field.

20.8 Compiled Functions

A compiled function has a data type of $$DTP-COMPILED-FUNCTION (001100). The pointer field points to a memory structure consisting of a $$DTP-COMPILED-FUNCTION-HEADER (101110) whose pointer field contains the number of instructions in the function. Following the header are six more words whose purposes are described below.

The code pointer CP is computed from the PC according to the rule given in the Program Counter chapter ($CP = 2 \times PC + 2^{25}$).


32 31 26 25 0
1 | 001100 | Pointer (P) to Compiled Function Header
31 26 25 0
P → 101110 | Number of Instructions in Function (N)
P+1 → Data Type | Compiled Function Name
P+2 → Data Type | Compiled Function Entry Points
P+3 → Data Type | Compiled Function Local Refs
P+4 → Data Type | Compiled Function Refs
P+5 → Data Type | Compiled Function Length
P+6 → Data Type | Compiled Function Code Pointer (PC)
31 26 25 0
CP−2 → 001100 | Back Pointer to Compiled Function (P)
CP−1 → #x 7FFFFFFE
CP → First Instruction, Low Word
CP+1 → First Instruction, High Word
... More Instructions...
CP+2N−2 → Last Instruction, Low Word
CP+2N−1 → Last Instruction, High Word

20.8.1 Compiled Function Name

This is the first word after the function header. It is a symbol, the name of the function.

20.8.2 Compiled Function Entry Points

This is the second word after the function header. It is a simple array containing the entry point information.

All even locations in the vector contain the number of arguments required to use this entry point. The odd locations contain the offset to the PC for the corresponding entry point. There will be more than one entry point only if optional arguments are used by the function.

When the function takes a @l[&rest] argument, then the last even location contains a negative fixnum. If you subtract this number from -1 you will get the minimum number of arguments required to use this entry point.

20.8.3 Compiled Function Local Refs

This is the third word after the function header. It is a simple array containing the local reference information. This is used to adjust the branch instructions inside the function when loading it.

The local references are stored as pairs. The first location of each pair contains the number of instructions to offset to the one to be adjusted.

The second location of the pair contains the number of instructions into the function that the branch is to.

20.8.4 Compiled Function Refs

This is the fourth word after the function header. It is a simple array containing the external reference information.

Each entry in the vector is a triple. The first word of the triple contains the number of arguments that the function is calling with. The second word of the triple contains the instruction offset to the call instruction. The third word contains a symbol pointer to the function being referenced.

20.8.5 Compiled Function Length

This is the fifth word after the function header. It is a fixnum, the number of instructions in the function. This is redundant with the information in the function header word.

20.8.6 Compiled Function Code Pointer

This is the sixth word after the function header. It has data type $$DTP-CODE and a PC in the pointer field. It points to the first instruction of the code before entry point offsetting.

20.8.7 Instruction Back Pointer

The instruction just before the one that the code pointer points to contains a special illegal instruction. The top 32 bits of this instruction are the "magic number" #x7FFFFFE and the low word contains a compiled function pointer back to the compiled function structure for this function.

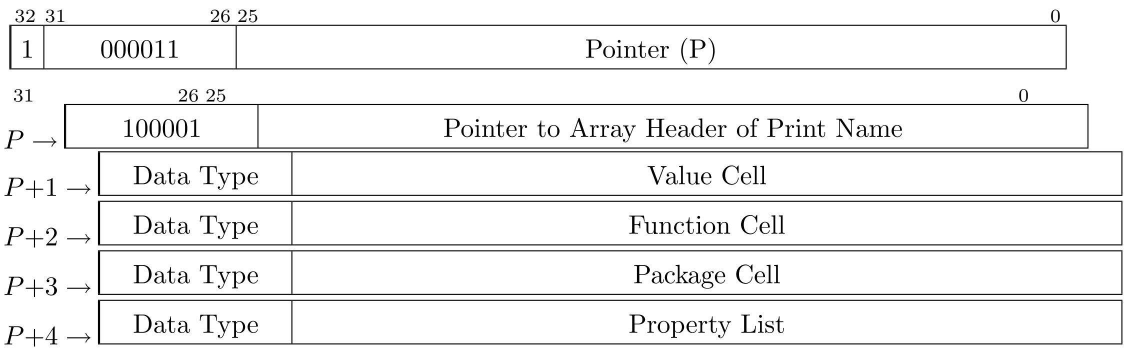
This is used to convert from a PC to a function pointer by scanning backwards in memory from the PC until the illegal instruction is found, and then following the pointer.

20.8.8 Changes to Format

The starting-address field has been eliminated; it and the code field were redundant.

20.9 Symbols

A symbol pointer has the data type $$DTP-SYMBOL (000011). Its pointer field points to a symbol structure in memory. The symbol structure contains five words, which are described below.



20.9.1 Symbol Header

The first word of a symbol structure has the data type $$DTP-SYMBOL-HEADER. Its pointer field points to an array header of an array containing the print name of the symbol.

20.9.2 Symbol Value

The second word of a symbol structure contains the value of the symbol. This can be either a visible Lisp object or the special unbound object. The unbound object is a word with a data type of $$DTP-UNBOUND whose pointer field points to the beginning of the symbol structure.

20.9.3 Symbol Function

The third word of a symbol structure (the "function cell") contains the function information for the symbol. Although it may contain any Lisp value, only the following are valid:

Unbound

The function cell contains the unbound object. The symbol has no function or macro definition.

Compiled Function

The function cell contains an object whose data type is $$DTP-COMPILED-FUNCTION.

Interpreted Function

The function cell contains either an object whose data type is $$DTP-INTERPRETER-CLOSURE or a list whose CAR is the symbol LISP:LAMBDA.

Compiled Macro

The function cell contains a cons cell whose CAR is the symbol LISP:MACRO and whose CDR is a compiled function.

Interpreted Macro

The function cell contains a cons cell whose CAR is the symbol LISP:MACRO and whose CDR is either an interpreter closure or a list whose CAR is the symbol LISP:LAMBDA.

20.9.4 Symbol Package

The fourth word of a symbol structure is the symbol's package. During the cold and warm loads, before the package system is installed, this word is a string which names the package. After the package system is installed, this word is the actual package object.

20.9.5 Symbol Property List

The fifth word of the symbol strucure contains the property list of the symbol.

20.10 NIL

NIL is a word whose data type is $$DTP-NIL (000000) and whose pointer field also contains all zeros. This makes it easy to test for NIL. Location zero in memory contains a degenerate symbol structure for NIL. All of its words are normal, except for the header. Both the header and value cell contain NIL (so CAR and CDR of NIL give NIL). The SYMBOL-NAME function knows this and handles the print name of NIL specially.

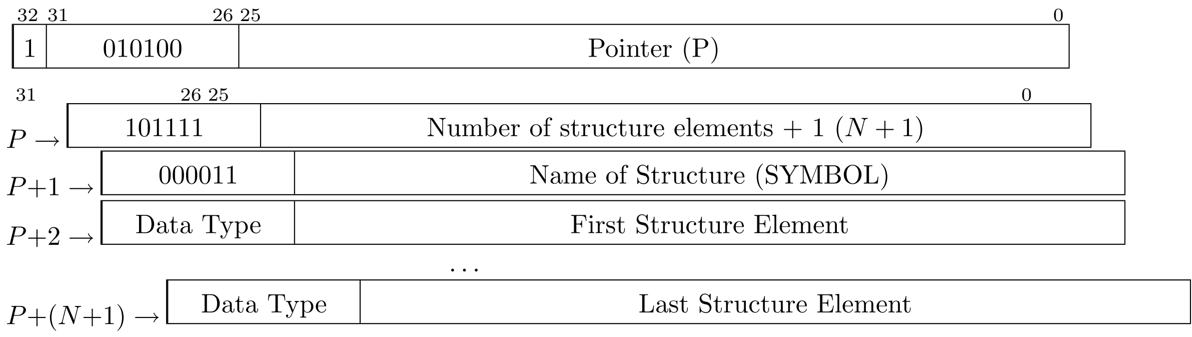
32	31 26	25 0
1	000000	00 00000000 00000000 00000000

20.11 T

T is a normal symbol. The only thing odd about it is that it has an absolute location of five, and the SET function won't let you change its value.

20.12 Defstruct Structure Instances

A structure instance has the data type $$DTP-STRUCTURE. The pointer field points to a memory structure introduced by a structure header. The structure header has data type $$DTP-STRUCTURE-HEADER (101111). Its pointer field contains the number of words in the structure, plus one for the name. The first word after the header is a symbol, the name of the structure. The subsequent words are the structure elements.



20.13 Undocumented So Far

Storage conventions for the following visible data types are either not documented or not established.

LOCATIVE, UNBOXED-LOCATIVE, CODE, STACK-GROUP, INSTANCE, LEXICAL-CLOSURE, INTERPRETER-CLOSURE, LEXICAL-ENVIRONMENT, EXTEND, ENCAPSULATION, HASH-TABLE.

Appendix A

List of Registers and Signals

Abbreviation	*Size (bits)*	*Name/Function*	*Section*
AF	8	Active Frame	9.2.3
HP	8	Heap Pointer	9.2.2
IR	64	Instruction Register	6.2
MCR	32	Memory Control Register	14.2
MD	32	Memory Data	15.1
MSR	32	Memory Status Register	14.3
OF	8	Open Frame	9.2.3
OREG	33	ALU Output	11
PCINC	24	Incremented Program Counter	7.3.2
RF	8	Return Frame	9.2.3
VMA	32	Virtual Memory Address	15.1

Appendix B

Functional I/O

This appendix contains tables and descriptions of functional sources and destinations.

B.1 Table of Functional Sources

Value	*Abbreviation*	*Meaning*
110100X	VMA	VMA Register
110101X	MD	MD Register
1100000	MEMORY-MAP	Memory Maps
1100001	GC-RAM	GC Ram
1100010	MEMORY-CONTROL	Memory Board Control Register
1100011	MICROSECOND-CLOCK	Microsecond Clock
1100100	???	Statistics Counter
1100101	TRAP-REGISTER	Trap Register
1100110	MEMORY-STATUS	Memory Board Status Register
1001000	PROCESSOR-STATUS	Processor Board Status Register
1001001	PROCESSOR-CONTROL	Processor Board Control Register
1001010	OPEN-ACTIVE-RETURN	Call Hardware: O, A, and R Registers
1001011	RETURN-PC-RETURN-DEST	Call Stack (RPC, and Return Dest)
1001100	CALL-HP-SP	Call Hardware: HP and CSP
1001101	TRAP-PC	Trap PC - (PC Two Clocks Earlier)
1001110	TRAP-PC+	Trap PC Plus - (PC One Clock Earlier)
1000111	ICACHE-A-HI	Read Cache A - Hi Data

1000110	ICACHE-B-HI	Read Cache B - Hi Data
1000101	ICACHE-A-LO	Read Cache A - Lo Data
1000100	ICACHE-B-LO	Read Cache B - Lo Data
1000011	TRAP-OFF	Read Trap Enable and Disable

B.2 Table of Functional Destinations

Value	*Abbreviation*
1000000	NOP, GUARANTEED TO DO NOTHING
1000001	RETURN-DESTINATION
1000010	NOP, WITH NO OVERFLOW TRAP (FOR COMPARES)
???????	RETURN
???????	RETURN-MV
1111000	VMA-START-READ-EARLY-NO-TRANSPORT
1111001	VMA-START-READ-EARLY
1111010	VMA-START-READ-EARLY-VISIBLE-EVCP
1111011	VMA-START-READ-EARLY-WILL-WRITE
1111100	VMA-START-READ-EARLY-CDR-NO-TRANSPORT
1111101	VMA-START-READ-EARLY-CDR
1111110	VMA-START-READ-EARLY-CDR-VISIBLE-EVCP
1111111	VMA-START-READ-EARLY-CDR-WILL-WRITE
1110000	VMA-START-READ-NO-TRANSPORT
1110001	VMA-START-READ
1110010	VMA-START-READ-VISIBLE-EVCP
1110011	VMA-START-READ-WILL-WRITE
1110100	VMA-START-READ-CDR-NO-TRANSPORT
1110101	VMA-START-READ-CDR
1110110	VMA-START-READ-CDR-VISIBLE-EVCP
1110111	VMA-START-READ-CDR-WILL-WRITE
1101100	VMA-START-WRITE-NO-GC-TRAP
1101101	VMA-START-WRITE
1101110	MD-START-WRITE-NO-GC-TRAP
1101111	MD-START-WRITE
110100X	VMA
110101X	MD *(write)*
1100000	MEMORY-MAP
1100001	GC-RAM
1100010	MEMORY-CONTROL
1100011	MICROSECOND-CLOCK
1100100	STATISTICS-COUNTER
1100101	TRANSPORTER-RAM
1001000	DATATYPE-RAM-WRITE-PULSE
1001001	PROCESSOR-CONTROL
1001010	OPEN-ACTIVE-RETURN
1001011	RETURN-PC-RETURN-DEST

1001100	CALL-HP-SP

B.3 Functional I/O Encodings

B.3.1 Processor Status Register

The PSR is read-only.

Bit(s)	*Meaning*
31:19	Undefined
18	Processor ALU_BOXED bit
17	Processor D_JUMP bit (active low)
16	Processor JUMP bit (active low)
15:13	Undefined
12:9	Return Destination Immediate
8:4	FPU Status Outputs
3:0	ECO jumper number

B.3.2 Processor Control Register

The PCR may be read or written to. All bits are zeroed by reset.

Bit(s)	*Meaning*
31:24	Unimplemented
23:20	Undefined
19	Floating Point Trap Enable (0 = disabled, 1 = enabled)
18	Call Hardware Stack Overflow Trap Enable (0 = disabled, 1 = enabled)
17	Call Stack Load Control (0 = normal, 1 = call stack special load sources)
16:13	Stack Group Number (selects one of 16 call hardware heap/stack groups)
12:9	PCTL_MISC, 4-bit field used for random things
8	PCTL_DATA bit, 1-bit data source for some special functions
7	HALT bit (0 = run, 1 = halt processor)
6	Box Mux Mode (0 = normal, 1 = register reload mode)
5	Floating Point Status RAM Write Enable (0 = read, 1 = write)
4	J_INDIR, indirect jump bit selectable by jump condition
3	Spare
2	ICACHE_ZENBL, low core cache set (0 = reset/disable, 1 = enable)
1	ICACHE_BENBL, B cache set (0 = reset/disable, 1 = enable)
0	ICACHE_AENBL, A cache set (0 = reset/disable, 1 = enable)

B.3.3 Memory Status Register

The MSR is read-only.

Bit(s)	*Meaning*
31:24	Undefined
23	Amount of memory installed (0 = 32 Mbytes, 1 = 16 Mbytes)
22	Autoboot jumper (0 = mastership external, 1 = boot automatically)
21	Memory Parity Error (0 = error, 1 = no error)
20:19	Undefined
18	MD Transport Trap (1 = MD read will cause transporter trap)
17	MD Page Trap (1 = MD read will cause read fault trap)
16	VMA Boxedness (0 = boxed, 1 = unboxed)
15	MD Boxedness (0 = boxed, 1 = unboxed)
14:13	Last Memory Cycle Type (see below)
12	Last Memory Cycle Type (0 = write, 1 = read)
11	Undefined
10:8	Nubus Bootstrap Mode (0 = normal, 1 = short reset, 2:7 = software)
7:4	ECO Jumper Number
3:0	Nubus Slot ID

Bits 14:13 —- 00 = will write, 01 = no evcp, 10 = transport, 11 = no transport. See the Transporter RAM chapter for details.

B.3.4 Memory Control Register

The MCR is read-write. All bits are zeroed by a reset.

Bit(s)	*Meaning*
31	Master Trap Disable (0 = no trapping, 1 = trap under other masks) See below.
30	Asynchronous Trap Enable (0 = disable, 1 = enable)
29	Overflow Trap Enable (0 = disable, 1 = enable)
28	Data Type Trap Enable (0 = disable, 1 = enable)
27	Synchronous Trap Enable (0 = disable, 1 = enable)
26	Single step on trap exit (0 = disabled, 1 = enabled)
25	Spare
24	Reset Trap Bit (0 = reset trap bit on, 1 = normal)
23:20	Undefined
19	DRAM Parity Error Flagging (0 = disable, 1 = enable)
18	Boot PROM (0 = enabled, 1 = disabled)
17:16	Transporter RAM Mode Select
15	Use L or C valid/write-enable bits in map (0 = C bits, 1 = L bits)
14	Write Wrong Parity to DRAM (0 = normal parity, 1 = wrong parity)

13	16384 microsecond interrupt (0 = disable/reset request, 1 = enable)
12	1024 microsecond interrupt (0 = disable/reset request, 1 = enable)
11	I-Cache error clear (0 = disable/reset icache error traps, 1 = enable)
10:9	NuBus AD(1:0) bits for transfers
8	NuBus TM0 bit for transfers
7	LED 2 (0 = lit, 1 = unlit)
6	LED 1 (0 = lit, 1 = unlit)
5	LED 0 (0 = lit, 1 = unlit)
4	Statistics Source Polarity (0 = true, 1 = invert)
3:1	Statistics Counter Source (options listed below)
0	Statistics Counter Mode (0 = edge trigger, 1 = duration)

Bit 31 (Master Trap Disable) also sets/resets during trap exit/entry

Bits 3:1 determine the statistics counter source:

Value	*Source*
000	I-cache hit
001	Processor memory cycle
010	Instruction Status Bit
011	Undefined
100	PC in high core
101	Undefined
110	Undefined
111	Undefined

B.3.5 GC/Transporter RAM

This functional destination is read/write.

Bit(s)	*Meaning*
7	Transporter: Box Error
6	Transporter: Trap if not Oldspace
5	Transporter: Trap if Oldspace
4	Transporter: Trappable Pointer
3	GC: Quantum is in oldspace
2:0	GC: Quantum volatility

B.3.6 Trap Register

The trap register is read-only. It is loaded either on a trap or on a write to the Memory Control Register.

Bit(s)	*Meaning*
31	Reset
30	Single step trace trap
29	Instruction Cache - Parity Error
28	Instruction Cache - NuBus Error
27	Instruction Cache - NuBus Timeout
26	Instruction Cache - Map Fault
25	Processor Memory Read - Parity Error
24	Processor Memory Read - NuBus Error
23	Processor Memory Read - NuBus Timeout
22	Processor Memory Read - Map Fault
21	Processor Memory Read - Transport
20	Processor Memory Write - NuBus Error
19	Processor Memory Write - NuBus Timeout
18	Processor Memory Write - GC RAM
17	Processor Memory Write - Map Fault
16	Floating Point
15	Call Stack Overflow
14	Spare (zero)
13	Data Type
12	29332 ALU Overflow
11	Spare (zero)
10	NuBus Interrupt 7
9	NuBus Interrupt 6
8	NuBus Interrupt 5
7	NuBus Interrupt 4
6	NuBus Interrupt 3
5	NuBus Interrupt 2
4	NuBus Interrupt 1
3	NuBus Interrupt 0
2	1024 Microsecond trap
1	16384 Microsecond trap
0	Always zero

B.3.7 Memory Map

May be read or written. The Map RAM address is bits 25:10 of VMA.

Bit(s)	*Meaning*

31:12	Physical Address
11:8	Software Definable
7	Page in on-board memory
6:4	Page Volatility
3	Write Enable (C)
2	Valid (C)
1	Write Enable (L)
0	Valid (L)

B.3.8 Call Hardware (Open/Active/Return)

May be read or written.

Bit(s)	*Meaning*
31:24	Unimplemented
23:16	Open Register
15:8	Active Register
7:0	Return Register

B.3.9 Call Hardware (Return PC, Return Destination)

Read-only.

Bit(s)	*Meaning*
31	Spare
30:24	Return Destination
23:0	Return PC

B.3.10 Call Hardware Pointers (CSP, HP)

Read-write.

Bit(s)	*Meaning*
31:16	Unimplemented
15:8	Heap Pointer (HP)
7:0	Call Stack Pointer (CSP)

Contents