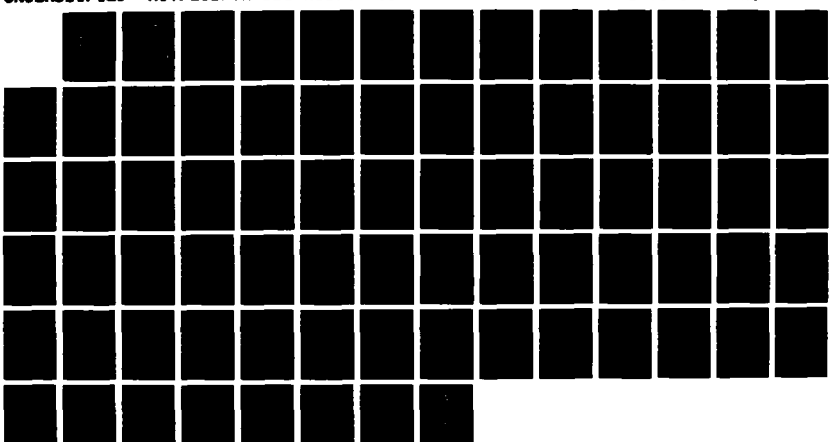
AD-A187 034 A MULTITHREADED PROCESSOR ARCHITECTURE FOR PARALLEL SYMBOLIC COMPUTATION(U) MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE.. T FUJITA SEP 87 1/1

UNCLASSIFIED MIT/LCS/TM-338 N00014-83-K-0125 F/G 12/6 NL



LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-338

A MULTITHREADED PROCESSOR ARCHITECTURE FOR PARALLEL SYMBOLIC COMPUTATION

Tetsuya Fujita

September 1987

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

ADA187034

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TM-338	5 MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125 and N00014-84-K-0099
6a NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b OFFICE SYMBOL (If applicable)	7a NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
A MULTITHREADED PROCESSOR ARCHITECTURE FOR PARALLEL SYMBOLIC COMPUTATION

12 PERSONAL AUTHOR(S)
Fujita, Tetsuya

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
Technical	FROM ____ TO ____	September 1987	71

16 SUPPLEMENTARY NOTATION

17 COSATI CODES		
FIELD	GROUP	SUB-GROUP

18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Futures, multithreaded architecture, parallel computing, tagged architecture

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper describes the Multilisp Architecture for Symbolic Applications (MASA), which is a multithreaded processor architecture for parallel symbolic computation with various features intended for effective Multilisp program execution. The principal mechanisms exploited for this processor are multiple contexts, interleaved pipeline execution from separate instruction streams, and synchronization based on a bit in each memory cell. The tagged architecture approach is taken for Lisp program execution and trap conditions are provided for future object manipulation and garbage collection.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b TELEPHONE (Include Area Code) (617) 253-5894 22c OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

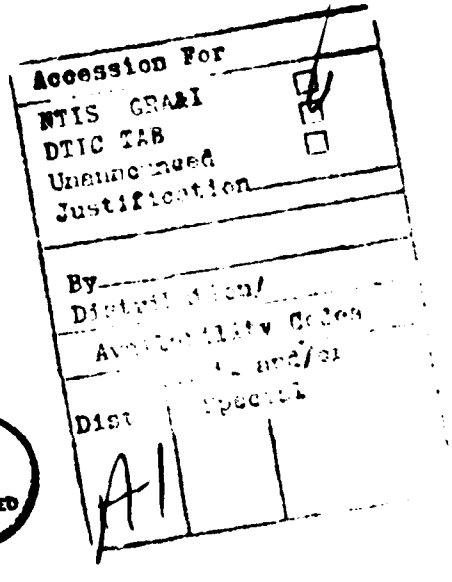
Unclassified

A Multithreaded Processor Architecture for Parallel Symbolic Computation

by

Tetsuya Fujita
NEC Corporation, Japan
Visiting Scientist in Laboratory for Computer Science

Massachusetts Institute of Technology
26 August. 1987



Abstract

This paper describes the Multilisp Architecture for Symbolic Applications (MASA), which is a multithreaded processor architecture for parallel symbolic computation with various features intended for effective Multilisp program execution. The principal mechanisms exploited for this processor are multiple contexts, interleaved pipeline execution from separate instruction streams, and synchronization based on a bit in each memory cell. The tagged architecture approach is taken for Lisp program execution and trap conditions are provided for future object manipulation and garbage collection.

Keywords: Futures, multithreaded architecture, parallel computing, tagged architecture

This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under contract numbers N0001-83-K-0125 and N0001-84-K-0099.

Contents

List of Figures

List of Tables

1 Introduction

The motivations for designing the Multilisp Architecture for Symbolic Applications (MASA) come from a paper on concurrent Lisp machines [4]. This paper identifies requirements or challenges to architects of concurrent Lisp machines and proposes a processor architecture. This architecture has multiple register sets to reduce the cost of procedure invocation, process creation, and context switch.

We borrowed ideas or concepts from HEP-1 [6,10] and SPUR [11]. We decided to take the RISC approach because we wanted to get a picture of the design framework of the processor early in the project and also make its structure as simple as possible. HEP-1 inspired us to apply ideas of multiple contexts, synchronization based on the empty/full bit of each memory cell, and the mechanism for feeding instructions from separate independent streams into the pipeline.

MASA has several features for Multilisp [2] program execution. Future objects are treated as a data type and the trap mechanism can detect touching of future objects. Since future objects are as common as usual Lisp objects in Multilisp programs, we provide special hardware support for fast trap handling. MASA is a tagged architecture and has several architectural features to support garbage collection algorithms based on generation scavenging [7,12] and incremental garbage collection [1,8] techniques.

In the rest of this paper, we describe MASA by starting with data types and memory format in Section 2. Section 3 discusses the design philosophy behind the MASA instruction set. Then we provide the general hardware overview of the MASA processor and the functional descriptions for its building units. The next two sections describe architectural issues concerned with tasks and traps in detail. Section 7 provides a complete description of the MASA instruction set and Section 8 explores its special features by means of some example programs. Finally, Section 9 concludes and mentions possible future work and extensions.

2 Data Types and Memory Format

This section describes the data types, the memory word organization, and the data representations for the MASA processor.

2.1 Data Types

From the hardware point of view, there are five distinct classes of data types. They are FIXNUM, NIL, CONS, FUTURE and all other (software defined) data types. FIXNUM and NIL are immediate types, and the others are pointer types. Each data object has a type tag field and a data field. In immediate types, the data field contains the value of the object, and the type tag field indicates the type of this value. In pointer types, the data field is interpreted as the address of some object in memory, and the type tag field states the type of this referenced object.

Table 1 shows a proposed definition of data types and tag values. The tag field is six bits long and the most significant bit is used to distinguish between immediate and pointer types. The first four types are treated specially by the hardware. Trap conditions are provided for detecting these types. In this scheme, we can introduce two kinds of software defined data types. The software defined immediate type is assigned the tag value of 01xxxx. This kind of type is treated as an immediate type for the EQL comparison, but is excluded for FIXNUM arithmetic operations. CHARACTER is a candidate for this type. The software defined pointer type is assigned the tag value of 11xxxx. It is interpreted as a pointer all the time just like CONS and FUTURE.

The differences from Multilisp [3] data types are:

- TASK, LABEL, LFILE, HUNK, CODE, CPOINTER, INSTRUCTIONS, OPERATOR, and PC are excluded because they are all closely related to the C implementation of MCODE.
- EXCEPTION is excluded because it can be represented as a particular kind of undetermined future object.
- RATIO, COMPLEX, FORWARDING and MISC-TYPE are added.

Tag code	Type name	Hardware detectable?	Immediate or Pointer?
000xxx	NIL	yes	immediate
001xxx	FIXNUM	yes	immediate
100xxx	CONS	yes	pointer
101xxx	FUTURE	yes	pointer
11xxxx	SYMBOL	no	pointer
	STRING	no	pointer
	VECTOR	no	pointer
	ARRAY	no	pointer
	STRUCTURE	no	pointer
	ENV	no	pointer
	CLOSURE	no	pointer
	FLONUM	no	pointer
	BIGNUM	no	pointer
	RATIO	no	pointer
	COMPLEX	no	pointer
	FORWARDING	no	pointer
	MISC-TYPE	no	pointer

Table 1: Data types

2.2 Memory Word Organization

Each memory word consists of two flag bits, two tag fields, and one data field. Figure 1 shows this structure. The *Empty/Full bit (EFbit)* tells whether or not the data field contains some value. This bit is used for concurrent task synchronization. Load instructions can only read "full" objects. A trap occurs when a task attempts to read an "empty" object. A task can lock an object by intentionally setting its EFbit empty. The *Old/New bit (ONbit)* allows an incremental garbage collection algorithm to tell whether the data field has a pointer to the old space or the new space. A transport trap occurs when a task tries to load a value with the ONbit old.

The generation tag field is intended for generation scavenging garbage collection. It is two bits long and therefore four generation categories are defined. In this field, generation 0 means the oldest generation and generation 3 means the newest generation. A generation trap occurs when a pointer to be stored belongs to a younger generation than the pointer to the area where it is to be stored; in other words, when a younger pointer is stored into an older object. The type field is six bits long and contains the type tag value mentioned in Section 2.1. The data field is thirty-two bits long and holds an immediate value or a pointer address.

When a memory word is loaded into the processor, only the generation tag, the type tag, and the data fields are copied into a register. General purpose registers have these three fields as shown Figure 2, and special registers such as a program counter or a status word register have only a data field.

2.3 Data Representations

Lisp data objects are represented in one word or more than one word. In the latter case, contiguous words are allocated to an object. The first word of a variable length object is a FIXNUM giving the length of the object. Figure 3 shows some typical Lisp object representations in the MASA.

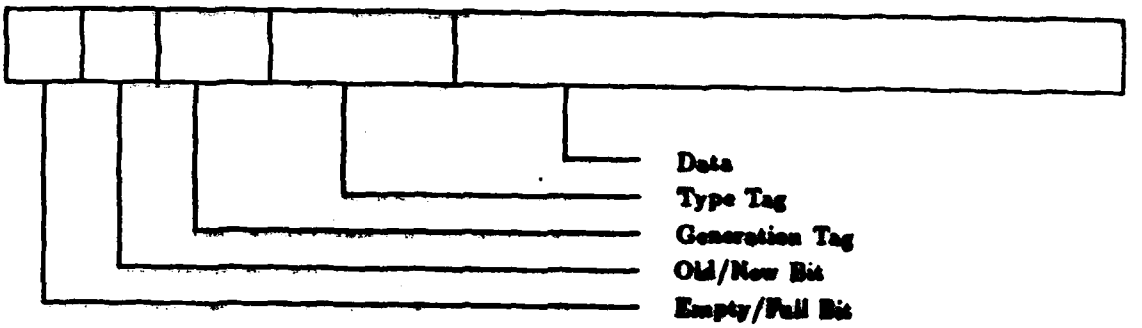


Figure 1: Memory Word Format

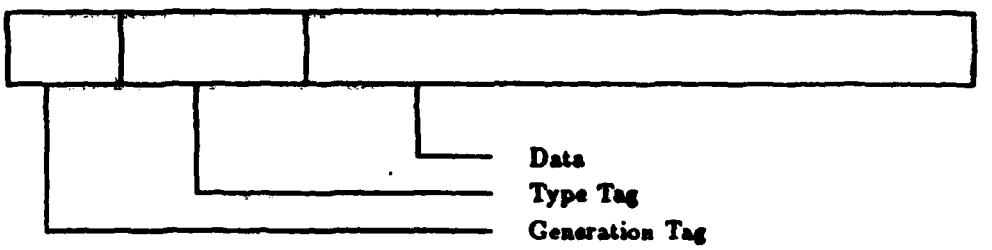


Figure 2: General Purpose Register

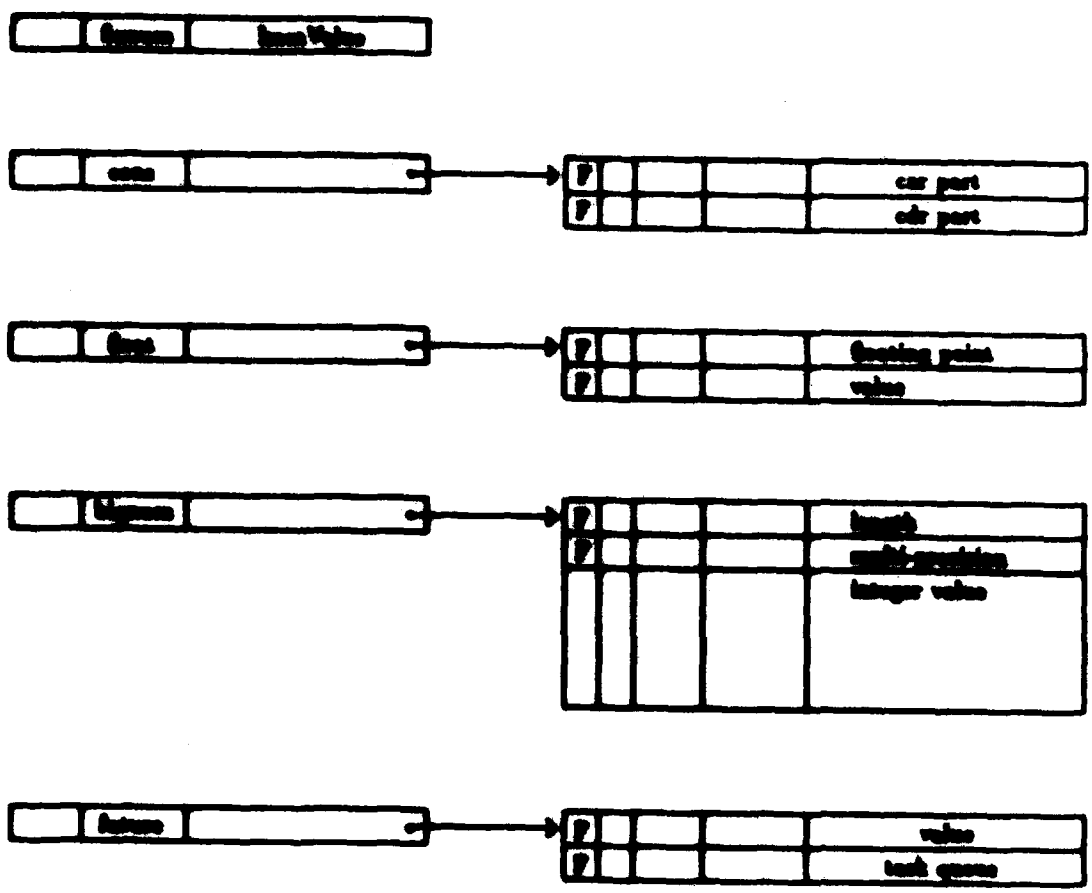


Figure 3: Data Representation

3 Instruction Set Design Philosophy

In this section we describe design philosophy lying behind the MASA instruction set design.

3.1 Multiple Contexts

MASA accommodates multiple contexts and allows us to switch among these contexts essentially without any overhead. We neither have to save nor restore any processor state at a context switch and we can pass arguments or results between tasks directly without accessing memory.

In this multiple context scheme, procedure call, future invocation, and trap handling create a new task that gets its own context and starts execution in this context. A task can access registers in its parent and child task contexts as well as in its own private context using register-to-register instructions like the following.

```
ADD     CHILD.R1,R2,R3
ADD     R1,PARENT.R2,PARENT.R3
```

The first instruction adds R2 and R3 in the private context and stores the result into R1 in the child context. The second instruction adds R2 and R3 in the parent context and stores the result into R1 in the private context. As this example shows, instructions can specify an operand register using a pair consisting of a context identifier and a register name. The context identifier can be left out if the register belongs to the private context.

3.2 Task Frames

In MASA, a task has its own context and a context is represented by a *task frame*, which consists of a set of task status registers and a set of general purpose registers. Each task in a processor is identified by a task frame number. Figure 4 shows the main components of a task frame.

Task status registers include a program counter, a status word, a child task register, a parent task register, and a trap information register. The

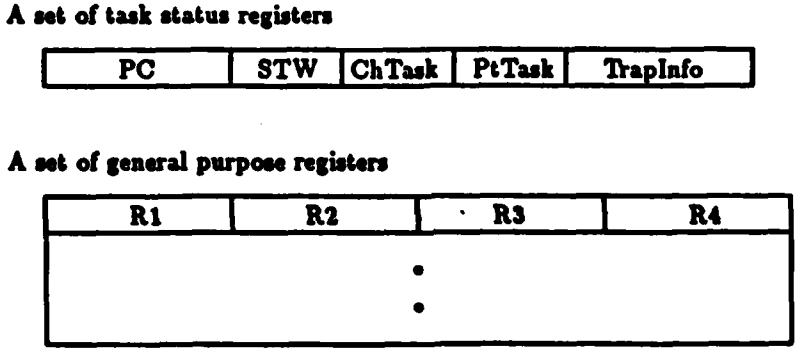


Figure 4: Task Frame

status word register has condition flags such as zero, negative, carry and overflow, and a task state bit. A task that is allocated to a task frame has only two states,*ready* and *running*. The ready state means that the task is ready to execute the next instruction pointed to by the program counter. The running state means that an instruction for this task is being issued and not yet completed or this task is waiting for some other task's termination.

The child task register (ChTask) holds the child task frame number and the parent task register (PtTask) holds the parent task frame number. For a procedure call, the child task is the callee and the parent task is the caller. For trap handling, the child task is the trap handler and the parent task is the task which caused the trap. A task can access a register in its child or parent task frame by referring to these two registers.

The trap information register (TrapInfo) provides source and destination register names of the trapped instruction. The TrapInfo consists of three fields: SR1, SR2, and DST. Trap handlers can get information from and store back to the trapped task frame using these fields like the following example.

```
ADD     R1,TRAPPED.SR1,TRAPPED.SR2
ADD     TRAPPED.DST,R1,R2
```

The first instruction adds the first and second source registers of the trapped instruction and stores the result into the R1 in the private task frame. The second instruction adds R1 and R2 in the private task frame and stores the result back to the destination register of the trapped instruction. We use TRAPPED instead of PARENT when SR1, SR2, or DST is specified as a register name.

Every register in each task frame is identified by a pair consisting of a task number and a register number. This two dimensional addressing scheme for the multiple register set is different from the one dimensional scheme applied to the register window in the Berkeley RISC processors [9,5]. We employ the two dimensional model since adjacent windows are not guaranteed to be allocated to consecutive window requests from a single control thread in the concurrent task environment. Since a task is limited to access only its child and parent tasks, this model is still simple at the

instruction set level.

3.3 Tagged Architecture

MASA takes a tagged architecture approach to dealing with the generic operations and garbage collection characteristic of Lisp program execution. As we describe in Section 2.2, every memory word has four distinct tag fields: the EFbit, the ONbit, the generation tag, and the type tag fields. Only the last two ones come into the processor.

Some trap conditions are based on tag checking results. "Addressing modes" for register operands can be specified, allowing instructions to extract the generation or the type tag field before data operations. Thus we can apply comparison or computation operations to these tag fields, just like the data field, without explicitly extracting them by another instruction in advance. The following is an example of field extraction.

```
CMPR    R1.TYPE,FIXNUM-TAG
ADD     R1,R2.GEN,1
```

The first instruction compares R1's type tag field with the constant FIXNUM-TAG. The condition flags in the status word register are updated based on this comparison. The second instruction increments R2's generation tag field and stores the result into the R1.

Once a tag field is extracted into the lowest bits of a register, we can also make a multiway branch based on a tag value by specifying the register as the offset to the target address in a branch instruction. The following is an example of a multiway branch.

```
BRRG    ALWS,PC,R2.TYPE
```

This instruction unconditionally branches to the target address calculated by adding the type tag field of R2 to the program counter. In other words, this instruction makes a PC-relative multiway branch based on the type tag field. In order to update tag fields of a general purpose register, we have two kinds of tag insertion instructions.

```
INST    R1,R2,FIXNUM-TAG
INSG    R3,R4,OLDEST-GENERATION-VALUE
```

The first instruction replaces R1's type tag field with the constant FIXNUM and R1's generation tag and data fields with R2's generation tag and data fields. The second instruction replaces R3's generation tag field with the constant OLDEST-GENERATION-VALUE and R3's type tag and data fields with R4's corresponding fields. The addressing modes for register operands are described in detail in Section 7.1.

3.4 Procedure Call/Return

A procedure obtains a new task frame when it is called. The caller task frame, like a frame on an activation stack, keeps its context until the callee task returns control so that we don't have to save the caller's state such as the program counter or the status word. A typical procedure call protocol is the following:

1. Request a new task frame to be allocated for the procedure invocation.
2. Transfer arguments to this child task frame.
3. Transfer control to the frame and wait for its termination.
4. After the termination, get results from the child task frame.
5. Finally release the task frame.

The following is an example of procedure call and return.

```
CALLER:     RQFR
            ADD       CHILD.R1,R2,0,MOVE
            ADD       CHILD.R2,R5,0,MOVE
            CALL      PROC1
            ADD       R2,CHILD.R3,0,MOVE
            RLFR      CHILD
            :
PROC1:      ADD       R3,R1,R2
            RETN
```

The RQFR instruction requests a task frame for the execution of the procedure PROC1. ADD instructions are used just for moving data from one register to another. The second and third instructions transfer R2 and R5 in the current task frame into R1 and R2 in the child task frame. The procedure PROC1 adds the two arguments, stores the result into R3, and then returns control to the caller task. After that, the caller task takes the result from the child task frame and then releases it by the RLFR instruction. Section 7 describes the operation of these instructions in detail.

For concurrent task activation, we need the same first two steps as in the above protocol. The next step we have to take is just making the new task "ready" and keeping the original task executing:

1. Request a new task frame allocation for the task activation.
2. Transfer data to this child task frame.
3. Activate the child task and still continue executing.
4. Possibly synchronize with the child task.

The following is an example of activating a concurrent task.

```
TASK1       RQFR
            ADD       CHILD.R1,R2,0,MOVE
            ACTK      TASK2
            :
```

The ACTK makes the task TASK2 become ready and still continues TASK1's execution. ACTK is described in detail in Section 7.

3.5 Traps

MASA has several trap conditions concerned with data types, synchronization, garbage collection, and some exceptional conditions. For example, an ADD instruction causes a trap when either of the two operands is not a FIXNUM number. This instruction performs a simple addition of two integer numbers in one instruction and leaves more complicated cases such

as addition of a FIXNUM and a floating point number to a trap service routine. This is our basic approach to generic operations. The trap mechanisms check frequently examined but seldom met conditions. Section 6 describes trap conditions in detail.

When a trap condition is detected, a trap token is generated and travels through the pipeline. A new task frame is allocated to a trap handler by hardware in the last pipeline stage. Some pieces of information conveyed by a trap token are stored into this task frame. They include the entry address of a trap handler, the source and destination register names of the trapped instruction, and the effective address for memory operation traps. Each trap condition has its own trap handler routine with a fixed entry point address.

If there is only one task frame available when a trap occurs, another kind of trap occurs and the occurrence of the original trap is remembered in a special register called the *suspended frame requests register (SFR)*. The service routine for this trap, called a *frame saver*, unloads some task frames to make space and then examines the SFR. One bit of the SFR is associated with each task frame and is set if trap processing for the corresponding task is suspended due to shortage of task frames. The frame saver re-executes the trapped instruction in each of these suspended tasks at the end of its processing.

4 General Organization

In this section we describe the general hardware organization of the MASA processor.

4.1 Pipeline Structure

Figure 5 shows the general organization block diagram of the MASA processor. It has a four-stage pipeline structure. The four pipeline stages are instruction fetch, operand fetch, operation execution, and register write stages as shown in Figure 6.

The fetched instruction word is stored in the stage1 register, and the fetched operands are stored in the stage2 register for the ALU/Shifter inputs. For store instructions, the stage2 register holds three operands; two for the address calculation and one for the write data. In this case, one of the address calculation operands is immediate data which is taken from an instruction word, and thus the register file needs only two output ports for the register read access.

There are two kinds of stage3 registers. The output of the ALU/Shifter is stored into the stage3 Load/Store Unit (LSU) register as the effective address for load/store instructions. For the other instructions, the output of the ALU/Shifter is stored into the stage3 ALU register. If any trap condition is detected in the Tag Checker and Generator, a trap token is stored into the stage3 ALU register instead of a data token. There is a direct data path from the stage2 register to the stage3 LSU register. The write data travels along this path and is stored into a part of the stage3 LSU register. Instructions change the contents of the Task Database and the Register File at the end of the fourth stage. Figure 7 shows the formats of the data tokens from stage1 through stage3.

4.2 Instruction Fetch Unit

The Instruction Fetch Unit selects one of the ready tasks in the Task Database and issues an instruction fetch to the Instruction Cache. Since

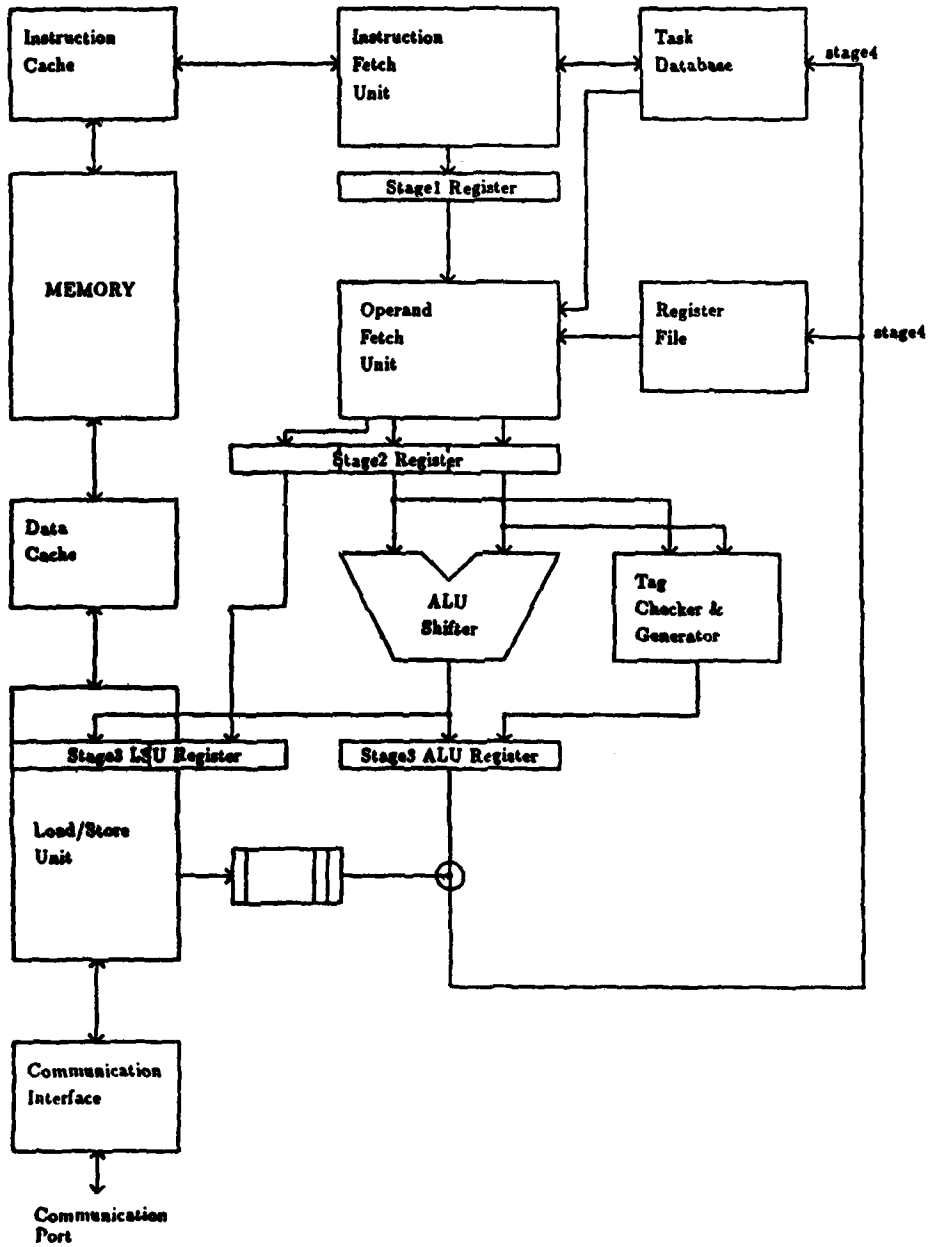


Figure 5: General Organization Diagram

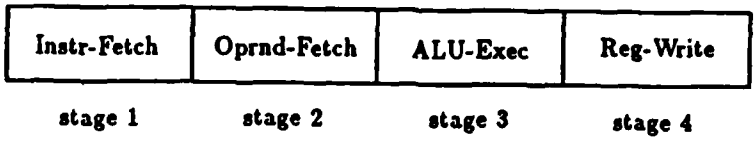


Figure 6: Pipeline Stages

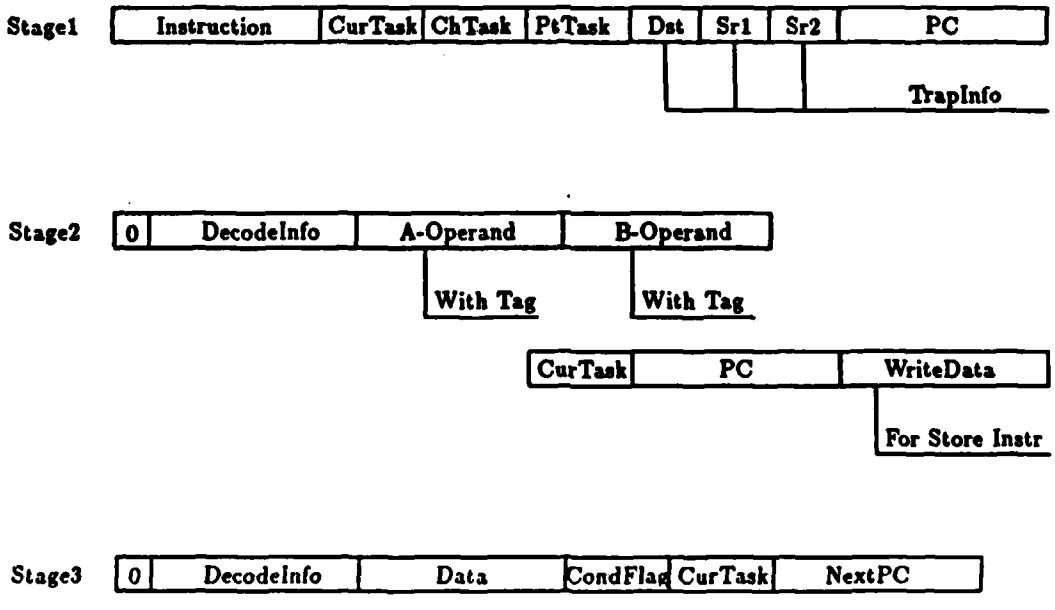


Figure 7: Data Token Formats

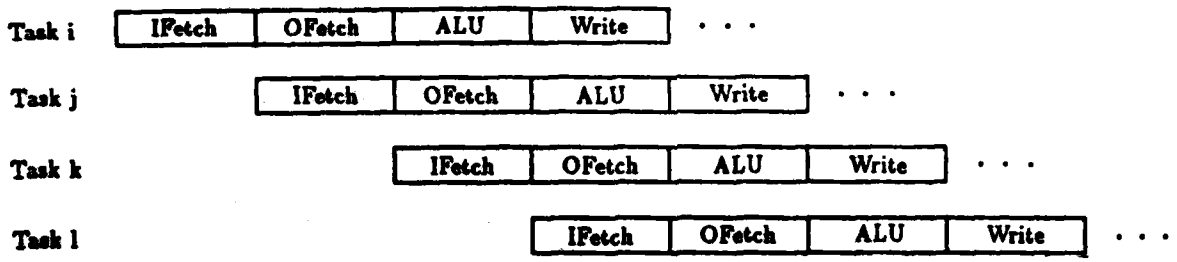


Figure 8: Multiple Task Execution

the Task Database can hold multiple tasks, instructions are fed into the pipeline from separate instruction streams in a multithreaded way. Each information token in the stage registers has a task frame number; thus context switching can be performed at every pipeline cycle on an instruction-by-instruction basis. This fast context switching mechanism increases tolerance of memory or communication latency in the multiprocessor system by offering the pipeline slots to other ready tasks while putting the waiting ones aside. Filling the pipeline with instructions from the independent threads makes the pipeline structure simple because no special hardware mechanisms are necessary for detecting pipeline hazards or resolving data dependencies. Figure 8 shows how multiple tasks are executed in the pipeline.

The Instruction Fetch Unit reads the PC and also some other status registers from the Task Database in the first pipeline stage. They include the ChTask, PtTask and TrapInfo registers and are stored in the stage1 register. In the next pipeline stage the Operand Fetch Unit will use these registers for accessing child or parent task frames. The Instruction Fetch Unit updates the task status bit of the status word register from the ready

to the running state. This is the only state transition performed at stages other than the fourth stage.

4.3 Operand Fetch Unit

The Operand Fetch Unit decodes operand portions of the instruction word in the stage1 register, and fetches or creates operand data in the stage2 register. The operand data comes from the Task Database, the Register File, or the instruction code in the stage1 register.

A register is specified in the instruction by a pair consisting of a task frame name such as private or child and a register name such as R1 or PC. This pair is called a *relative register identifier*. A register in the Task Database or the Register File, on the other hand, is identified by a pair of a task number and a register number. This pair is called an *absolute register identifier*.

The Operand Fetch Unit performs the translation from relative register identifiers to the absolute register identifiers based on the ChTask, PtTask and TrapInfo fields in the stage1 register. Figure 9 shows the hardware unit for this register specifier translation. The Operand Fetch Unit includes three translator units for two source and one destination register specifiers. A child task frame is translated into the task number in the ChTask field of the stage1 register, and similarly a parent task frame is translated into the task number in the PtTask field of the stage1 register. A trapped task frame is translated by using the PtTask and the TrapInfo in the stage1 register.

Each operand data token in the stage2 register has a generation tag, a type tag, and a data field like the general purpose registers. The Operand Fetch Unit may extract the type or the generation tag field into the data field before storing in the stage2 register if the instruction specifies operations to these fields. A tag value is put in the lowest end of the data field. It is extended with zeros and the generation 0 and FIXNUM tag values are put in the tag field.

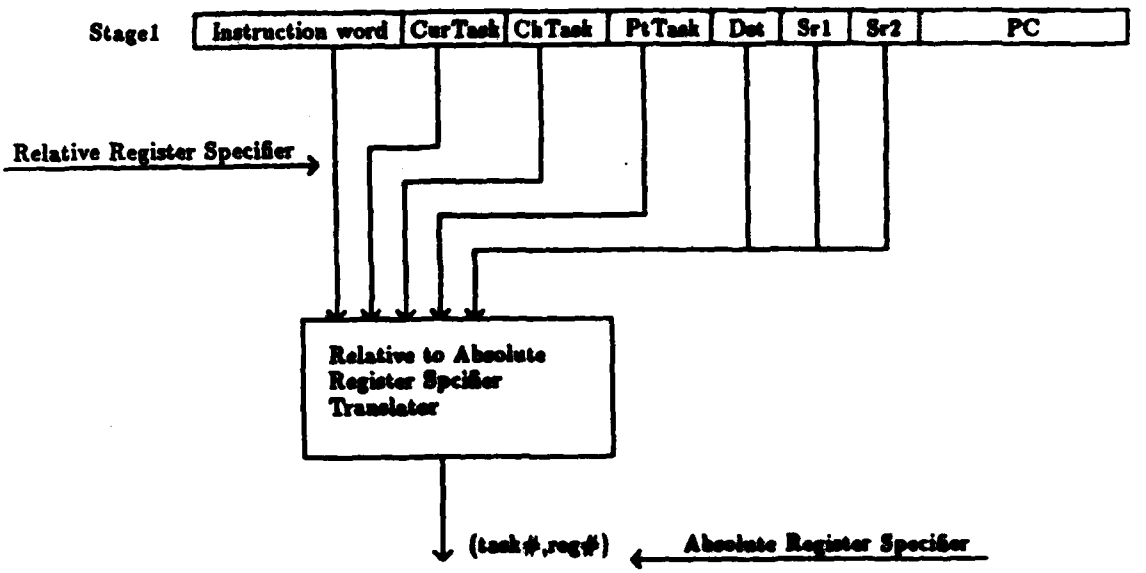


Figure 9: Relative to Absolute Register Specifier Translator

PC	STW	ChTask	PtTask	TrapInfo
PC	STW	ChTask	PtTask	TrapInfo
⋮	⋮	⋮	⋮	⋮
PC	STW	ChTask	PtTask	TrapInfo

Figure 10: Task Database

4.4 Task Database and Register File

The Task Database consists of several sets of task status registers and the Register File consists of several register sets. Each register in both register banks is accessed with an absolute register identifier. Figures 10 and 11 show the structure of the Task Database and the Register File. The description of status registers is found in Section 5.

4.5 Load/Store Unit

The Load/Store Unit issues memory read and write operations to the Data Cache and the Communication Interface. When a memory operation is finished, the Load/Store Unit sends a token to the Task Database and the Register File. For load instructions, this token has a task number, data, and a destination register specifier, and the data is stored into the destination register in the corresponding task frame in the fourth pipeline stage. The task state bit of the task frame is also updated from running to ready at this point of time. For store instructions, a token has a task number and

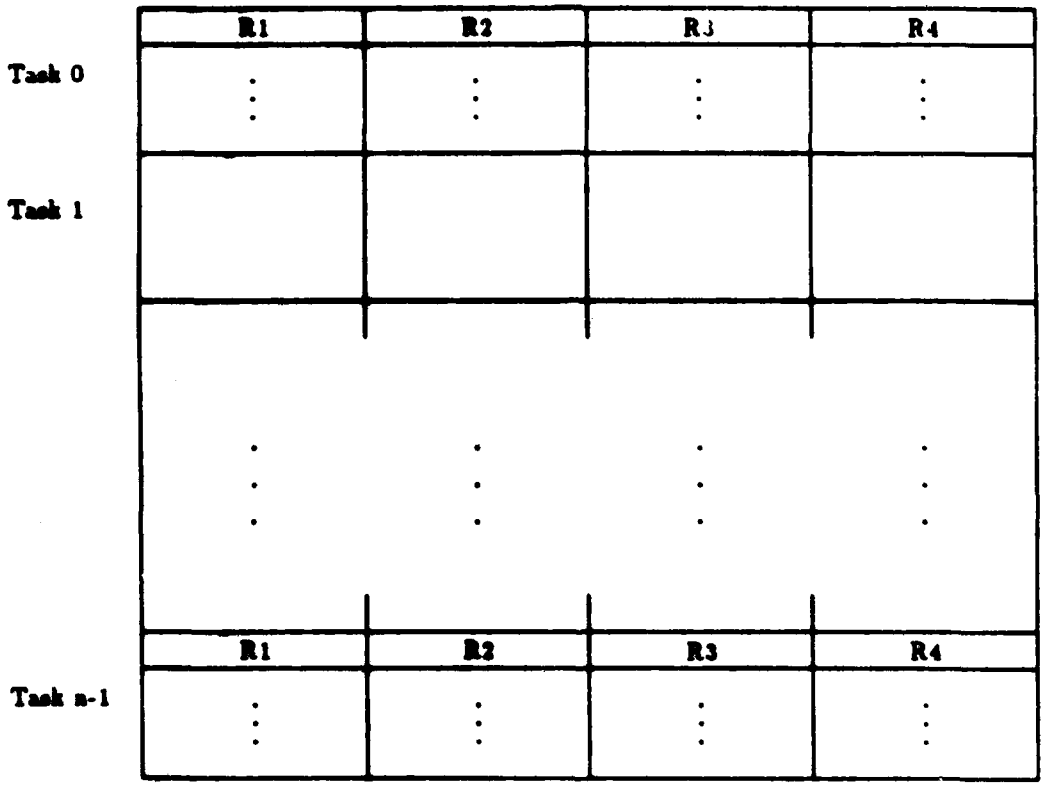


Figure 11: Register File

the task state bit is updated in the fourth stage.

The Load/Store Unit detects the Transport, Generation, Empty Location, and Full Location trap conditions. This unit generates trap tokens when these traps occur. The trap token includes the effective address of the location being accessed at the time of the trap: this information is stored into the task frame which is allocated to the trap's handler.

4.6 Tag Checker and Generator

The Tag Checker and Generator checks tag fields of the two ALU operands for trap conditions and generates the tag field of the output data token stored in the stage3 ALU register. Type tag fields are compared with particular tag values such as FUTURE, CONS, NIL, or FIXNUM, and also compared with each other for equality.

The Tag Checker and Generator can also take the lowest-order data bits from the second ALU operand and puts them in the tag field of the ALU output data. The tag field in the ALU output data can come from three sources: the same value as the first operand's tag, some fixed value such as generation 0 or FIXNUM, and the lowest-order bits of the second operand data.

5 Task Representation

In MASA, each task has its own set of task status registers and general purpose registers. It keeps its own context and is executed along with other concurrent tasks on the processor. The Task Database consists of multiple sets of task status registers and the Register File consists of multiple sets of general purpose registers.

5.1 Task Status Registers

Task status is a set of task status registers assigned to a task. It includes several special registers such as a program counter or a status word register which are essential for program flow control. Table 2 describes each of these task status registers.

The PC is updated at the end of the last stage of a pipeline. The value of the PC is conveyed along the pipeline and is incremented to form the next PC in some stage before the last. When a trap occurs, the PC in the trap-causing task frame is not updated and holds the address of the trapped instruction so that a trap handler can easily re-execute the same instruction after trap processing. In procedure invocation, the PC in the caller side is incremented and retains the return point. Thus a task frame keeps the information that is pushed onto the activation stack in a single context processor.

An allocated task frame has two states, ready and running. The task state bit represents these two states. A task becomes ready when the PC points to the next instruction to be executed, and enters the running state when the fetched instruction is sent into the pipeline. In the MASA implementation, the transition from ready to running is performed at the end of the first instruction fetch stage and the transition from running to ready is done at the end of the last write stage. A task which is trapped or waiting for termination of a called procedure stays in the running state until the control is returned to it. After termination, the waiting task becomes ready and its execution restarts with the instruction pointed to by the PC.

Register Name	Meaning
PC	Program Counter This register is updated at the last stage of the pipeline with the incremented PC, the branch target address, or the trap handler entry address.
STW	Status Word This register includes condition flags (ZNVC: see Figure 12) and the task state bit (TSbit).
ChTask	Child Task Number This register contains the task number allocated at procedure invocation. Some instructions can read from or write to registers in the child task frame by referring to this register in a register specifier.
PtTask	Parent Task Number This register holds the resident/nonresident bit (RN-bit) and the task number allocated to the parent task. The parent task states the caller task at procedure invocation and the trapped task in trap handling.
TrapInfo	Trap Information This register has three parts: DST, SR1, and SR2. They hold information on the destination and the two source register specifications in the trapped instruction.

Table 2: Task Status Registers

A task number is set into the ChTask register when a new task frame is assigned as a result of the RQFR instruction. The task number in the PtTask register has two meanings. When the CALL instruction is executed, the caller task number is set into the PtTask register in the callee task frame. When a trap occurs, the trap-causing task number is set into the PtTask register in the trap handler's task frame.

The resident/nonresident bit in the PtTask register tells whether the parent task frame is resident or not resident. When a processor runs short of task frames, it causes the Task Frame Unavailable trap and activates the frame saver to unload inactivate task frames and make enough space. The frame saver sets the resident/nonresident bit of a task frame when it unloads its parent task frame. The RETN instruction checks the resident/nonresident bit and causes the Return to Saved Task trap if the bit turns out to be set. When a task attempts to access its parent task frame, it has to explicitly check the resident/nonresident bit to make sure the parent frame is still resident on the processor.

5.2 Task Frame Management

An available task frame is allocated when the RQFR instruction is executed or a trap occurs. It is deallocated or released by the RLFR or the RERL instructions. A task frame has three states. It is free when not allocated to some task. After it is allocated, it can switch between ready and running in the course of instruction execution.

The *task frame usage register (TUR)* keeps the allocation status for each task frame. Each bit is associated with a task and updated by hardware when a frame is allocated or deallocated.

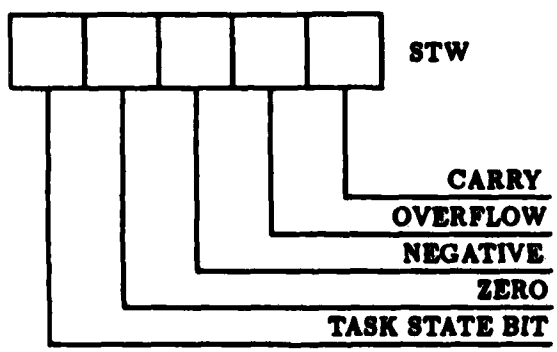


Figure 12: Status Word Register

6 Traps

Each trap condition is associated with the entry address of its own trap handler; the hardware can generate the entry point address from the information a trap token conveys. Trap conditions are detected by hardware so that we can handle simple cases quickly without paying attention to infrequent complicated cases. This scheme helps deal with generic operations and tag manipulation

6.1 Trap Mechanism

When a trap condition is detected, a trap token is generated and passed through the pipeline. Each trap token has the entry address of the trap handler, register specifier information of the instruction being executed, and the effective address for traps relevant to memory operations. Figure 13 shows the format of trap tokens.

In the last (write) stage of the pipeline, a new task frame is allocated to the trap handler by hardware. All the write operations to the trapped task frame are suppressed so that it can keep the state just before the trap occurred. Instead, the trap handler's task frame receives the information from the trap token. The entry address is stored into the trap handler's PC, the source and destination information is saved into the trap handler's TrapInfo register, and the effective address is written into the R1 in the trap handler's register set. The trap handler's PtTask register now has the trapped task number and the task state bit is set to "ready" to execute the first instruction of the trap handler program.

A trap handler resumes execution of a trapped task in two ways: by re-executing the same instruction that caused the trap, or by proceeding to the next instruction. For example, in the case of touching a determined future, the Future Touching trap handler re-executes the same instruction after loading the value cell of the future into the original source register. On the other hand, the Fixnum trap handler proceeds to the next instruction after storing the result into the original destination register. The PC in the trapped task frame retains the address of the trapped instruction, so

to re-execute the same instruction, it is only necessary to make the task ready. A trap handler increments its trapped task frame's PC in order to proceed to the next instruction.

6.2 Trap Conditions

Table 3 shows the trap conditions supported in MASA. In this table, the higher conditions have higher priority than the lower ones. For example, if the Task Frame Unavailable and Fixnum traps occur in the same instruction, the Task Frame Unavailable trap handler is activated.

Table 4 indicates which instructions enable which trap conditions. The detail on instructions is provided in Section 7.

6.3 How to Deal with Task Frame Unavailable Trap

The Task Frame Unavailable trap occurs when the last free task frame is about to be allocated to a new task at procedure invocation or trap handling time. If this happens, the last frame is not given to the original task but to the frame saver, which makes space by saving several inactive task frames out of the processor. The frame saver looks into the relationship between individual tasks and decides which task frames should be unloaded. It sets the resident/nonresident bit in the child task frame of each unloaded task.

Some additional frame requests may come before the frame saver has finished its job. These requests, including the first one, are recorded in a register called the *suspended frame requests register (SFR)* so that the frame saver can deal with suspended requests after it has made enough space. Each bit of the SFR corresponds to a task number and is set to one when the task asks for a task frame while the frame saver is working. This register can be accessed as a special register and its contents are reset automatically when it is read so that no requests can be written into the SFR between the read and the reset operations. The frame saver just sets these suspended tasks back to the ready state so that the instructions that made unsuccessful frame requests are retried under more auspicious circumstances.

Trap[1]	Detection	Description
Illegal Instruction (IL)	Instruction decode unit	Undefined instruction code is detected.
Task Frame Unavailable (TF)	Resource manager	The last free task frame is assigned.
Return to Saved Task (RS)	Resource manager	A procedure returns control to its caller which is not resident in the processor.
Future Touch (FT)	Tag checker	An instruction has tried to touch a future object.
EQL (EQL)	Tag checker	The two operands in an equality comparison are of the same pointer type but have different values.
Cons or Nil (CN)	Tag checker	The first operand of the address calculation is neither of cons nor nil type.
Cons (CS)	Tag checker	The first operand of the address calculation is not of cons type.
Fixnum (FX)	Tag checker	Either of two operands is not of fixnum type.
Overflow (VF)	ALU	An overflow condition is detected at ALU.
Software Trap (ST)	Instruction decode unit	A TRAP instruction is executed or a TRTG instruction is executed with tag mismatch.
Empty Location (EL)	Load/store unit	The accessed memory word is locked.
Full Location (FL)	Load/store unit	The accessed memory word is unlocked.
Transport (TR)	Load/store unit	A pointer to the old space is read from memory.
Generation (GE)	Load/store unit	The stored word is a pointer and the first operand of the address calculation is older than the word to be stored.

Notes:

1. Names within parentheses in this column are used throughout this document for designating particular trap conditions.

Table 3: Trap Conditions

Trap	Enabled for
Illegal Instruction (IL)	any instruction
Task Frame Unavailable (TF)	any instruction
Return to Saved Task (RS)	RETN
Future Touch (FT)	Data operation instructions including CMPR, ADD, SUB, AND, OR, XOR, and SHFT unless atrp disables "future touch" only for ADD and SUB or LDxx if ltrp enables "future touch."
EQL (EQ)	CMPR if cc = "eql."
Cons or Nil (CN)	LDxx if ltrp = "cons-or-nil."
Cons (CS)	LDxx if ltrp = "cons."
Fixnum (FX)	CMPR, ADD, SUB, AND, OR, XOR, and SHFT unless it is optionally disabled only for CMPR, ADD and SUB.
Overflow (VF)	ADD and SUB unless atrp disables "overflow."
Software Trap (ST)	TRAP and TRTG
Transport (TR)	LDxT
Empty Location (EL)	LDxx, or STxx if strp enables "empty."
Full Location (FL)	STxx if strp enables "full."
Generation (GE)	STxG

Table 4: Trap Enable Conditions

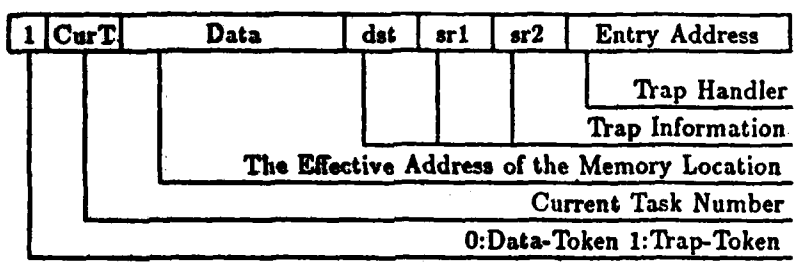


Figure 13: Trap Token Format

7 Instruction Set

The MASA instruction set is divided into five categories: data operation, load/store, task frame management, program control, and tag insertion instructions. MASA can be classified as a RISC architecture. It has a small number of instructions (actually twenty seven instructions), it has only load/store instructions for memory operations, and each instruction is executed in a single pipeline execution, that is, four pipeline stages, without microcode. The following sections describe the operation of the instructions, focusing on special issues such as task synchronization, task frame management, or tag manipulation. Table 5 gives mnemonics, operands, and brief description for all the instructions. Table 6 explains the notations used in operand descriptions of Table 5.

7.1 Register Specifiers

A register specifier is a portion of an instruction code and selects a certain register as a source or destination. The source register specifier consists of three parts: frame selection, register name, and field extraction control. The destination register specifier has only the first two parts. Figure 14 shows the structure of these register specifiers. The frame selection part specifies one of the three relevant task frames, that is, the private, the child, and the parent task frames. The register name part designates one of the registers in a task frame including both the general purpose registers and the task status registers. This field can also designate the source or destination registers of the trapped instruction by using trap information stored in the private task frame. Special registers such as the suspended frame requests register and the task frame usage register can also be specified in this field.

Decoding each source register specifier generates an operand data token for the ALU/Shifter. The data token has a generation tag, a type tag, and a data field. The field extraction part of a source register specifier controls these three fields. It can extract any of the three fields from the register specified by the first two parts of the source register specifier. Figure 15

Mnemonics	Operands	Meaning
CMPR	sr1,sr2i,cc	compare operands
ADD	dst,sr1,sr2i.atrp	addition
SUB	dst,sr1,sr2i.atrp	subtraction
AND	dst,sr1,sr2i	logical and
OR	dst,sr1,sr2i	logical or
XOR	dst,sr1,sr2i	logical exclusive or
SHFT	sc,dst,sr1	shift
LDFT	dst,sr1,sr2i,ltrp	load and set full with transport trap
LDET	dst,sr1,sr2i,ltrp	load and set empty with transport trap
LDFN	dst,sr1,sr2i,ltrp	load and set full without transport trap
LDEN	dst,sr1,sr2i,ltrp	load and set empty without transport trap
STNG	dst,imm,src,strp	store and set new with generation trap
STOG	dst,imm,src,strp	store and set old with generation trap
STNN	dst,imm,src,strp	store and set new without generation trap
STON	dst,imm,src,strp	store and set old without generation trap
RQFR		request task frame
RLFR	frm	release task frame
RERL		return and release task frame
CALL	addr	call procedure
ACTK	addr	activate task
RETN		return from procedure
BRAD	bc,addr	branch to immediate address
BRRG	bc,sr1,sr2i	branch to register address
TRAP		software trap
TRTG	sr1,sr2i	trap on tag mismatch
INST	dst,sr1,sr2i	insert type tag field
INSG	dst,sr1,sr2i	insert generation tag field

Table 5: Instruction Mnemonics and Operands

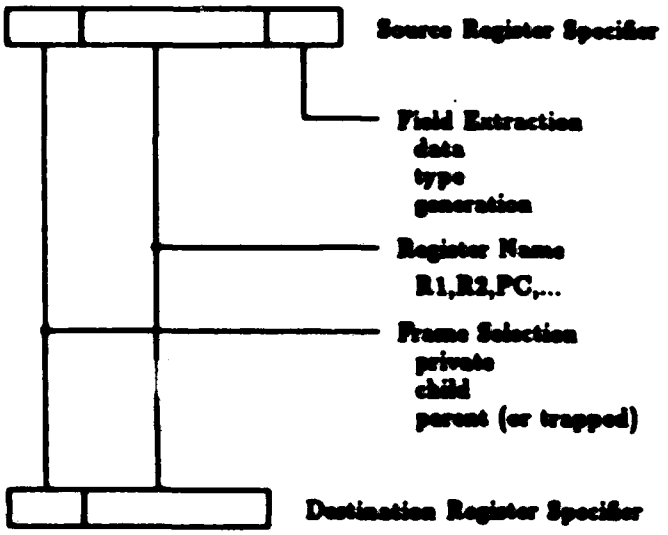


Figure 14: Register Specifier

Names	Meaning
dst	destination register specifier
sr1	1st source register specifier
sr2i	2nd source register specifier or immediate constant
imm	immediate constant
frm	task frame specifier (see Table 19 for detail)
addr	immediate target address
bc	branch condition (see Table 21 for detail)
cc	comparison code (see Table 10 for detail)
sc	shift operation control code (see Table 14 for detail)
ltrp	trap specifier for load instructions (see Table 16 for detail)
strp	trap specifier for store instructions (see Table 17 for detail)
atrp	trap specifier for arithmetic instructions (see Table 13 for detail)

Table 6: Notations for Operand Descriptions

shows the functions of the field extraction. As the first line of this figure shows, if the data field is specified in the field extraction field, no field extraction is performed and all the three fields stay in the same position in the operand token. (Task status registers and special registers have neither generation nor type tag field. They are treated as a FIXNUM of generation 0.) If the generation or the type tag field is selected in the field extraction as shown in the second and third lines in the figure, the tag field is extracted into the lowest bits of the data field in the operand data token. The leading remaining bits are filled with zeros. In these cases, the tag of the operand data token gets a FIXNUM of generation 0. The field extraction part comes only with the source register specifier. All of the three fields are updated at the same time when data is stored into a general purpose register. In a source register specifier, R0 means a FIXNUM zero with the oldest possible generation field.

The register specifier designating the source or destination register of the trapped instruction specifies the trapped task for the frame selection

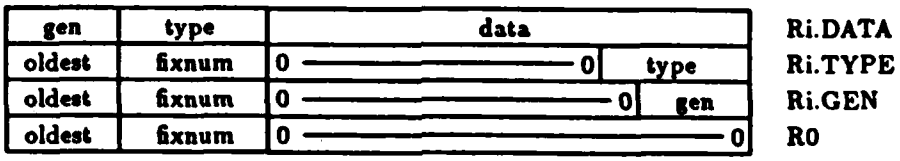


Figure 15: Field Extraction

and one of DST, SR1, or SR2 for the register name. This scheme is a kind of register indirect register addressing mode since the content of DST, SR1, or SR2 is interpreted as a register name.

Throughout this paper we use the following notation for a register specifier: `frame.reg.field` for a source register specifier and `frame.reg` for a destination register specifier. Tables 7 through 9 summarize the choices for `frame`, `reg`, and `field` parts. The `frame` part can be left out if it specifies PRIVATE and the `field` part can be omitted if it specifies DATA. Note that a trapped instruction register comes only with the TRAPPED frame selection, and a special register such as SFR and TUR is specified without the frame selection part. The following is some examples of the notation.

```
CHILD.R1.TYPE
PARENT.R2
R3.GEN
TRAPPED.SR1
```

Names	Meaning
PRIVATE	the current task's frame
CHILD	the child task's frame
PARENT or TRAPPED[1]	the parent task's frame

Notes:

1. We use both PARENT and TRAPPED for the parent task frame. Remind that the PtTask register holds the parent task number for task and procedure activation and the trapped task number for trap handling.

Table 7: Frame Selection

Names	Meaning
SFR	the suspended frame requests register
TUR	the task frame usage register
PC	the program counter
STW	the status word register
CHTASK	the child task number register
PTTASK	the parent task number register
TRAPINFO	the trap information register
Rx	a general purpose register
DST	the destination register of the trapped instruction
SR1	the first source register of the trapped instruction
SR2	the second source register of the trapped instruction

Table 8: Frame Selection

Names	Meaning
DATA	the data field
GEN	the generation tag field
TYPE	the type tag field

Table 9: Frame Selection

7.2 Data Operation Instructions

The comparison instruction CMPR updates four condition flags (ZNVC) in the STW register based on the result of comparing sr1 and sr2i. The comparison can be performed by interpreting the operands as either signed integers or as unsigned integers. CMPR enables the "Fixnum (FX)" or EQL trap conditions. Two FIXNUM numbers can be compared with each other by a single CMPR instruction with FX enabled. If FX is enabled, a trap will occur when either of the operands is not FIXNUM so that a trap handler can perform the comparison in software. CMPR with EQL enabled checks EQL equality between two objects. The data fields are compared with each other for the arithmetic comparison. The EQL comparison compares both the type tag and data fields. Condition flags are examined by conditional branch instructions. These comparison interpretation and trap enabling are specified by the cc operand of CMPR. Table 10 summarizes the choices for this operand.

Specifier	Meaning
	All the condition flags are updated based on signed arithmetic comparison. The data fields are compared. FX trap condition is enabled.
UNS	All the condition flags are updated based on unsigned arithmetic comparison. The data fields are compared. FX trap condition is enabled.
EQL	All the condition flags are updated based on signed arithmetic comparison. The data and type tag fields are compared. EQL trap condition is enabled.

Table 10: Comparison Code (cc)

We can compare two tag fields with each other or compare a tag field with a constant by exploiting tag field extraction for sr1 and sr2i in CMPR instructions.

```
EXAMPLE1:   CMPR    R1.TYPE,R2.TYPE
            BRAD    EQ,LABEL1
            :
EXAMPLE2:   CMPR    R1.TYPE,MAXIMUM-IMMEDIATE-TAG
            BRAD    GT,LABEL2
            :
```

In the second example we can check whether R1 contains an immediate or pointer type object using two instructions.

Arithmetic, logical and shift operation instructions perform their operations on the data fields of the operands. The tag for the arithmetic operation results is the same tag as in the first operand, sr1. The Future Touch (FT), Fixnum (FX), and Overflow (VF) trap conditions are enabled for usual arithmetic operations. VF and FT, however, may need to be disabled for explicit address calculations and all of FT, FX, and VF are disabled when ADD or SUB are used for moving operations by specifying zero data as the second operand. The atrp operand for ADD and SUB instructions serves for this trap disabling purpose. Table 11 describes the actions of data operation instructions and Table 12 shows the notations used in the instruction description tables. Table 13 shows possible choices for atrp operand.

Mnemomics	Operations
CMPR	Four condition flags (ZNVC) are updated according to the comparison results between sr1 and sr2i.
ADD SUB	dst ← sr1 op sr2i ZNVC flags are updated.
AND, OR XOR	dst ← sr1 op sr2i Z and N condition flags are updated.
SHFT	dst ← sr1 op sr2i Z, N, and C condition flags are updated.

Table 11: Data Operation Instructions Descriptions

Name	Meaning
EFbit	empty/full bit of a memory word
ONbit	old/new bit of a memory word
ChTask	child task register of a task status
TFU	task frame usage register
TaskState	task state bit of status word register
CurrentTask	current task number for which the instruction is executed
Dst,Sr1,Sr2	dst, sr1, and sr2 register numbers of the trapped instruction
Data	data field of a trap token
dst.gen	generation tag field of the destination register
dst.type	type tag field of the destination register
dst.data	data field of the destination register
op	arithmetic, logical or shift operator

Table 12: Notation for Instruction Descriptions

Specifier	Meaning
	Future touch, overflow, and fixnum trap conditions are enabled.
DSVF	Fixnum condition is enabled but overflow and future touch conditions are disabled.
MOVE	Future touch, overflow and fixnum are all disabled.

Table 13: Trap specifier for arithmetic instructions (atrp)

FT and FX trap conditions are always enabled for logical and shift operation instructions. Table 14 shows the four choices for the sc operand in the SHFT instruction. The difference between the arithmetically left and logically left shift operations is the arithmetical one would signal overflow while the logical one would ignore it. The arithmetically left shift is used for the software multiply calculation.

Name	Meaning
ARR	arithmetically right
LGR	logically right
ARL	arithmetically left
LGL	logically left

Table 14: Shift Operation Control Code (sc)

7.3 Load/Store Instructions

Load instructions have two distinctions: whether or not the transport trap condition is enabled, and whether the empty/full bit (EFbit) of the memory word read stays full or becomes empty after the read operation. Thus four load instructions are provided. Their operations are described in Table 15.

The transport trap occurs if the memory word to be accessed for read is a pointer and points to an object in the old space. A trap handler needs to read this old pointer without causing this trap when moving the object it points to from the old space to the new space. LDFT and LDET serve this purpose.

The EFbit is used for synchronizing concurrent accesses to a shared variable. Every load instruction to an empty memory word causes the Empty Location (EL) trap. Thus, LDEx instructions, which leave the accessed memory location empty, exclude other accesses to the location and can be used for the first read access of an atomic operation such as test-and-set.

Mnemomics	Operations
LDFx	dst ← M[sr1+sr2i] M[sr1+sr2i].EFbit ← "full" The difference between LDFT and LDFN is that LDFT enables but LDFN disables the transport trap condition.
LDEx	dst ← M[sr1+sr2i] M[sr1+sr2i].EFbit ← "empty" The difference between LDET and LDEN is that LDET enables but LDEN disables the transport trap condition.
STNx	M[dst+imm] ← src M[dst+imm].ONbit ← "new" M[dst+imm].EFbit ← "full" The difference between STNG and STNN is that STNG enables but STNN disables the generation trap condition.
STOx	M[dst+imm] ← src M[dst+imm].ONbit ← "old" M[dst+imm].EFbit ← "full" The difference between STOG and STON is that STOG enables but STON disables the generation trap condition.

Table 15: Load/Store Instructions Descriptions

Load instructions may cause three data-type-related traps in the the address calculation from sr1 and sr2i. They are Future Touch, Cons or Nil, and Cons trap conditions. CAR and CDR can be implemented by a single load instruction with the Cons or Nil trap enabled as follows.

```
CAR:        LDFT      R1,0(R2),CN+FT
CDR:        LDFT      R1,1(R2),CN+FT
```

The offset to the pointer (R2) is zero for CAR and one for CDR. The trap specifier `CN+FT` enables Future Touch and Cons or Nil traps. If the type tag of R2 says FUTURE, the Future Touch trap occurs. If its type tag is CONS or NIL, this load instruction issues a read operation without causing any type trap. If its type tag is neither FUTURE, CONS, nor NIL, this load instruction causes the Cons or Nil trap. Note that the Future Touch trap has higher priority than the Cons or Nil trap. The Cons trap condition is intended for list alteration operations such as RPLACA or SCAR. The ltrp operand of load instructions enables these kinds of trap conditions. Table 16 shows the choices for the ltrp operand.

Specifier	Meaning
	Future touch, cons-or-nil, and cons trap conditions are disabled.
FT	Future touch trap condition is enabled.
CS+FT	Cons and future touch trap conditions are enabled.
CN+FT	Cons-or-nil and future touch trap conditions are enabled.

Table 16: Trap Specifier for Load Instructions (ltrp)

Store instructions have also two distinctions: whether or not the Generation trap is enabled, and whether the old/new bit (ONbit) of the accessed memory word is set to old or new after the write operation. As in the case of load instructions, there are four store instructions: their operation is described in Table 15.

STOx instructions are intended for garbage collection. The incremental

garbage collector moves an object from the old space to the new space when an old pointer is encountered in the new space. Pointers within this transported object are set to be old because the objects they point to are still in the old space.

The Empty Location (EL) and Full Location (FL) trap conditions can be optionally enabled in store instructions. Recall that a shared variable is locked during an atomic operation by setting its EFbit to be empty using an LDEx instruction. Write operations at the conclusion of an atomic operation are performed by store instructions with FL enabled in order to make sure that the shared variable is still empty (locked). Store instructions with EL enabled will not update shared variables while locked. Proper synchronization in a producer-consumer relationship can be achieved if the consumer uses LDEx instructions and the producer enables the FL trap for its store instructions. All store instructions leave memory locations full after a write operation. This trap is enabled for a store operation used as a RPLACA. The strp operand specifies the enabled trap conditions for store instructions. Table 17 shows the choices for this operand.

Specifier	Meaning
	Both empty and full location trap conditions are disabled.
EL	Empty trap condition is enabled.
FL	Full trap condition is enabled.

Table 17: Trap Specifier for Store Instructions (strp)

7.4 Task Frame Management Instructions

A task frame is requested and allocated by the RQFR instruction and is released by the RLFR or RERL instructions. The RLFR instruction can release the private, child and parent task frames. RLFR for the child or parent task frames passes control to the following instruction of the current task, while RLFR for the private task frame terminates the current task

execution. RLFR for the private task frame can be used at the end of a task for computing the goal value of a future object. The RERL instruction also releases the private task frame but transfers control to the parent task. Table 18 describes the operation of these three instructions.

Mnemomics	Operations
RQFR	ChTask ← next available task frame number The corresponding bit of TFU is set to "allocated." This instruction will cause a task frame unavailable trap if there is only one available frame.
RLFR	The current, child or parent task's frame is released and the corresponding bit of TFU is reset to "unallocated."
RERL	parent-task.TaskState ← "ready" The current task frame is released and the corresponding bit of TFU is reset to "unallocated." The task state of the current task frame remains unchanged and does not become "ready" after this instruction.

Table 18: Task Frame Management Instruction Descriptions

A task frame allocated for procedure invocation is requested by RQFR and released by RLFR instructions. RQFR makes a new task frame allocated and sets its task number in the ChTask register of the private task frame. The frm operand tells private, child, or parent task frame to be released in RLFR as Table 19 shows.

A task frame for trap handling is allocated by hardware and released by an RERL instruction. A trap handler stores the result into the trapped task frame by specifying a register name in the TrapInfo as a destination. Then it transfers control to the trapped task and releases its own task frame using RERL. The following is an example of the Future Touching trap handler.

```
FTR-TOUCH:   LDFT      R1,0(TRAPPED.SR1)
             ADD       TRAPPED.SR1,R1,0,MOVE
             RERL
```

Specifier	Meaning
PRIVATE	the current task frame The current task frame is released and won't be "ready" any more.
CHILD	the child task frame
PARENT	the parent task frame

Table 19: Task Frame Specifier for RLFR (frm)

The first instruction specifies SR1 in the TrapInfo register as a source to read the first word of the future object. TRAPPED.SR1 means the source register in the trapped instruction. The second instruction stores the memory word read by the LDFT instruction into the source register in the trapped instruction. The third instruction finally returns control to the trapped task. Section 8 describes this trap handler in detail.

7.5 Program Control Instructions

There are two instructions for making a child task become ready. They are the CALL and ACTK instructions. CALL and ACTK set the task state bit of the child task to "ready" and put the target address into the PC in the child task frame. CALL also updates the current task PC so that it points to the following instruction. In this way the return address of a procedure call is stored in the PC of the caller task frame. CALL also stores the current task number into the PtTask register of the child task frame. Control is returned back through this linkage when a procedure terminates execution. The difference between CALL and ACTK is that CALL keeps the current task state as "running" (thus preventing the current task from continuing to execute), while ACTK continues on to the next instruction in the current task. A caller waits in a running state for its callee's termination. ACTK makes two tasks, the current and the child tasks, ready in a single instruction. Table 20 describes operation of the program control instructions.

Mnemomics	Operations
CALL	private-frame.PC ← NextPc child-frame.TaskState ← "ready" child-frame.PC ← addr child-frame.PtTask ← CurrentTask The PC of the private task frame is updated but its task state remains unchanged and waits in a "running" state for its child task's termination.
ACTK	private-frame.PC ← NextPc private-frame.TaskState ← "ready" child-frame.PC ← addr child-frame.TaskState ← "ready" Both private and child tasks become ready and their PC's are updated. This is unlike usual load/store or data operation instructions where only the private task proceeds to the next instruction.
RETN	parent-frame.TaskState ← "ready" The next instruction executed in this control thread is that pointed to by the parent task frame's PC.
BRxx	PC ← target address The target address is a portion of an instruction word (addr) for BRAD and is calculated from sr1 and sr2i for BRRG.
TRAP	A new task frame is assigned to a trap handler. new-frame.TaskState ← "ready" new-frame.PC ← entry address of the trap handler new-frame.PtTask ← CurrentTask new-frame.TrapInfo ← Dst, Sr1, and Sr2 of a trap token new-frame.r1 ← Data of a trap token The entry address of the trap handler is taken from the NextPc field of a trap token.
TRTG	If the type tag of sr1 is not equal to sr2i, software trap occurs. new-frame.TaskState ← "ready" new-frame.PC ← entry address of the trap handler new-frame.PtTask ← CurrentTask new-frame.TrapInfo ← Dst, Sr1 and Sr2 of a trap token new-frame.r1 ← Data of a trap token The entry address of the trap handler is taken from the NextPc field of a trap token.

Table 20: Program Control Instructions Descriptions

The return instruction RETN simply makes the parent task state ready, which returns control to the caller task. When the parent task has been unloaded from the processor by the frame saver, RETN causes the Return to Saved Task (RS) trap. The handler for this trap restores the saved parent task frame and then re-executes the RETN instruction. Recall that whether or not the parent task is unloaded is recorded in the resident/nonresident flag in the PtTask register.

MASA has two kinds of branch instructions: BRAD and BRRG. The target address for BRAD comes out directly from the instruction word, but the target address for BRRG is calculated from sr1 and sr2i. BRRG makes a PC-relative branch by specifying the PC as sr1 and the offset as sr2i. Both branch instructions can be conditional by specifying some branch condition for their bc operand. Table 21 shows the possible choices for this operand.

Specifier	Meaning
ALWS	always
EQ,ZE	equal to zero
NE	not equal to zero
GT	greater than
GE	greater than or equal to (positive)
LT	less than (negative)
LE	less than or equal to
VF	overflow
CR	carry

Table 21: Branch Conditions (bc)

A Software trap occurs when TRAP or TRTG is executed. A new task frame is allocated to handle the trap and becomes ready. The entry point address of the trap handler is stored into the PC of this new task frame. The trapped task frame number is saved in the PtTask register of this task frame. TRAP always causes a software trap, but TRTG causes one only when the type tag field of sr1 is not equal to the lowest bits of sr2i. TRTG

allows us to trap a bad data type with a single instruction.

7.6 Tag Insertion Instructions

The tag insertion instruction INST replaces the type tag field of the destination register with the lowest bits of sr2i and the generation tag and data fields of the destination register with the corresponding fields of sr1. INSG does the same kind of update to the generation tag. It replaces the generation tag field of the destination register with the lowest bits of sr2i and the rest of the destination register with the corresponding fields of sr1. No special tag extraction operations are needed because tag fields can be extracted using register specifiers in ordinary data operation instructions. Table 22 shows operations of the INST and INSG instructions.

Mnemomics	Operations
INST	dst.gen ← sr1.gen dst.type ← sr2i dst.data ← sr1.data
INSG	dst.gen ← sr2i dst.type ← sr1.type dst.data ← sr1.data

Table 22: Tag Insertion Instruction Descriptions

7.7 Summary

To summarize the instruction set description, Table 23 shows the trap enabling and disabling control for all the instructions. In this table, IL and TF trap conditions are omitted because they are always enabled.

Table 24 summarizes the state update operations for all the instructions. In the table, DST can be either a task status register or a general purpose register. The Task Database needs two address lines for write operations.

Instruction	RS	FT	EQL	CN	CS	FX	VF	ST	TR	EL	FL	GE
LDxT		*		*	*				o	o		
LDxN		*		*	*					o		
STxG										*	*	o
STxN										*	*	
Task Frame Instructions												
CALL,ACTK												
RETN	o											
BRxx												
TRxx								o				
INSx												
CMPR		o	*			*						
ADD,SUB		*				*	*					
logical		o				o						
SHFT		o				o	*					

Notations: BLANK:disabled o:enabled *:optionally enabled
Refer to Table 3 for trap condition descriptions and Table 5 for summary information on instructions.

Table 23: Which Trap Conditions Instructions Enable

One is for the current task's bookkeeping and the other is for data updating. They are named wrt1 and wrt2 in the table. Only the TSbit and the PC are updated for two different task frames in the same instruction. We need expensive hardware constructs for these registers. The write enable signal for each TSbit has to be controlled by wrt1 and wrt2 so that the TSbit is updated if either wrt1 or wrt2 addresses this task frame. The PC requires another expensive structure. Each PC needs its own multiplexer between the Data field and the NextPC field of the stage3 ALU register. For the CALL or ACTK instructions, the private task frame's PC gets data from the NextPC field and the child task frame's PC gets data from the Data field. We have to control the write enabling and the multiplexer selection by wrt1 and wrt2. All the other registers use only the wrt2 address line for their write enable control.

Instruction	DST	Private				Child/TrapH			Parent	TUR	wrt1	wrt2
		ts	pc	ch	cf	ts	pc	pa	ts			
Load	o	◇	▷								own	dst
Store	o	◇	▷								own	dst
RQFR		◇	▷	o						set	own	own
RLFR		◇	▷							reset	own	
RERL									◇	reset		pa
CALL			▷			◇	o	●			own	ch
ACTK		◇	▷			◇	o				own	ch
RETN									◇			pa
BRxx		◇	o								own	own
TRxx[1]						◇	▷	●				ch
INSx	o	◇	▷								own	dst
Data op.	o	◇	▷		*						own	dst

Notes:

1. **Some trap information is stored into a trap handler's task frame by TRAP or TRTG instructions.**
2. **Blanks in the wrt1 and wrt2 columns mean a "don't care" case.**
3. **Terminology:**
   DST: destination register
   dst: destination task number
   ts: task status bit in the STW register
   pc: PC register
   ch: ChTask register or child task number
   pa: PtTask register or parent task number
   cf: Condition flags in the STW register
   TUR: task frame usage register
4. Each column uses the following notations.
   o: Data field of the stage three token
   ▷: NextPc field of the stage three token
   ●: Current task number field of the stage three token
   *: Condition flag field of the stage three token
   ◇: Transition from "running" to "ready"

Table 24: Which Registers Instructions Update

8 Example Trap Handlers

In this section we show example trap handler programs in the MASA instruction set.

1. Future Touch

   Description An operand of a future-touching type instruction is of type FUTURE.

   Trap Information SR1: a register number which holds a future pointer

   Processing The trap handler reads the memory location containing the future's value and places this value in the original source register. If the future is still undetermined, another trap, "Empty Location (EL)," occurs during this read operation. In the case of an EL trap, the original task is suspended until the future resolves. Otherwise, the instruction which caused the future touch trap is re-executed immediately. SR1 in the TrapInfo of the trap handler's task frame is loaded with the register specifier that holds the pointer to the touched future object so that this trap handler can deal with both sr1 and sr2 cases.

   Code Suppose the first word in the future object is the value cell and its EFbit is used for distinguishing determined or undetermined. The main part of this trap handler is as follows.

```
FTR-TOUCH:   LDFT      R1,0(TRAPPED.SR1)
             ADD       TRAPPED.SR1,R1,0,MOVE
             RERL
```

   If we can specify the trapped task register for both the source and the destination in the LDFT instruction, we can combine the first two instructions above into one instruction like the following.

```
FTR-TOUCH2:  LDFT      TRAPPED.SR1,0(TRAPPED.SR1)
```

   In this trap processing, the future pointer in the trapped task's sr1 is replaced by the contents of the future's value cell. If this

future object is still undetermined, the Empty Location trap occurs in the LDFT instruction.

2. EQL

Description The two operands are of the same pointer type but have different data fields. This trap is enabled for comparing two operands in the EQL sense.

Trap Information SR1, SR2

Processing The trap handler gets both operands and compares them after suitable memory operations. It restarts the trapped task with the next instruction after updating its condition flags based on the recomparison. Since the trapped task PC still points to the trapped instruction, the trap handler has to explicitly increment the PC before making the trapped task ready.

Code The first portion which takes both operands from the trapped task frame:

```
ADD         R1,TRAPPED.SR1,0,MOVE
ADD         R2,TRAPPED.SR2,0,MOVE
```

The portion of code which returns the control to the instruction following the trapped instruction:

```
ADD         TRAPPED.PC,TRAPPED.PC,1,DSV
ADD         R3,TRAPPED.STW,0,MOVE
; update the trapped condition flags
ADD         TRAPPED.STW,R3,0,MOVE
RERL
```

3. Cons or Nil

Description The first ALU operand is neither of type CONS nor of type NIL. This trap condition is enabled for the address calculation part of CAR and CDR operations.

Trap Information SR1

Processing The trap handler proceeds to an error handler after taking relevant information from the trapped task frame.

4. Cons

Description The first ALU operand is not of type CONS. This trap condition is enabled for the address calculation of RPLACx and RPLACx-EQ operations.

Trap Information SR1

Processing The trap handler proceeds to an error handler after taking relevant information from the trapped task frame.

5. Empty Location

Description The EFbit of the accessed memory location indicates empty. All read operations and some write operations to shared variables require the memory location to be full before executing.

Trap Information SR1; the effective address is saved in R1 of the trap handler

Processing Tasks which cause this trap in the course of read operations are blocked and retried later, while cases where a write operation causes this trap are handled by an error routine. The trapped task frame holds the same state as before the trap occurred because any update, including incrementing the PC, is canceled in the trap operation of the pipeline. Thus the trap handler can retry the memory access instruction just by making the trapped task ready. This trap can also occur when an undetermined future object is touched. In this case, the empty location trap handler should enqueue this task to be waked up when the future object becomes determined. A future read can be distinguished from a non-future read by the pointer's type tag.

6. Full Location

Description The EFbit of the accessed memory location indicates full. Store instructions can optionally enable this trap.

Trap Information the effective address is saved in R1 of the trap handler

Processing The trap handler proceeds to an error handler after taking relevant information out of the trapped task frame.

7. Fixnum

Description Either of two operands is not of fixnum type.

Trap Information DST,SR1,SR2

Processing The trap handler should take the sr1 and sr2 operands of the trapped instruction, perform the proper arithmetic operation after necessary coercion operations, store the result into the dst register of the trapped task frame, and then proceed to the next instruction of the trapped task.

Code The source and the destination registers of the trapped task frame are specified in the ADD instructions.

```
FIXNUM:     ADD     R1,TRAPPED.SR1,0,MOVE
            ADD     R2,TRAPPED.SR2,0,MOVE
; After coercing and calculation of the result in R1
            ADD     TRAPPED.DST,R1,0,MOVE
            ADD     TRAPPED.PC,TRAPPED.PC,1,DSVF
            RERL
```

8. Overflow

Description An overflow condition is detected at the ALU.

Trap Information DST,SR1,SR2

Processing A trap handler recomputes the result in a more appropriate data representation (e.g., BIGNUM instead of FIXNUM), and stores it into the original destination register.

Code The same kinds of instructions as the Fixnum trap are used in this handler.

9. Task Frame Unavailable

Description The last free task frame is about to be allocated to a new task. This trap can occur when a new task frame is explicitly requested by a procedure or future call, or is allocated by hardware for a trap handler.

Processing The processor gets in a special state where additional frame requests are suspended. A trap handler, called the frame saver, unloads several task frames to make space for suspended frame-requesting tasks. Then it reads the suspended frame requests register (SFR) and sets all the suspended tasks back to the ready state. At the end of this processing, the processor returns to its normal operating mode.

Trap Information SFR

Code The frame saver somehow chooses task frames to be saved and stores them to a frame saving area in memory.

```
; Go along task linkage.
        ADD     R1,PARENT.PTTASK,0,MOVE
        CMPR    R1,NONRESIDENT-MINIMUM
```

Since the RNbit is the most significant bit of the PtTask register, we can examine whether the parent task is resident or not by comparing the PtTask register with the constant NONRESIDENT-MINIMUM.

10. **Return to Saved Task**

Description Control is returned to an already saved task frame. This condition can be detected when a procedure returns to its caller task using the RETN instruction.

Trap Information the pointer to the saved task frame

Processing The trap handler gets the pointer to the saved frame, asks for a new frame allocation, and restores the frame into this frame. Then it resets the RNbit in the PtTask register and stores the frame number into the PtTask of the trapped task frame. The trapped RETN instruction is re-executed after this processing. The table of the pointers to the saved task frame is maintained in memory by the frame saver.

Code The trap handler first restores the saved frame. Suppose R7 contains the pointer to the saved frame in the following example.

```
                ADD         R1,TRAPPED.R7,0,MOVE
                RQFR
; at this point chtask has a new task frame.
                LDFT        TRAPPED.R1,0(R1)
                LDFT        TRAPPED.R2,1(R1)
                LDFT        TRAPPED.R3,2(R1)
                :
; reset RNbit and store the new task frame number
                ADD         TRAPPED.PTTASK,CHTASK,0,MOVE
                RERL
```

11. Transport

Description An attempt is made to load a pointer to an object in the old space into a processor register.

Processing The trap handler first checks whether or not the accessed object in the old space has already been transported to the new space and contains a forwarding pointer to the new location. If the object holds a forwarding pointer, this new location address is copied into the locations holding the original old pointer and then the trapped load instruction is re-executed. If the object has not yet been transported to the new space, the trap handler transports the accessed object from the old space to the new space, leaves the forwarding pointer in the copy of the object in the old space, and then stores the pointer to the transported object into the locations holding the original old pointer and re-executes the trapped load instruction.

Trap Information the effective address is saved in R1 of the trap handler

Code The handler moves the object that caused the trap from the old space to the new space, and then stores the pointer to the

transported object into the originally accessed old pointer. The address of this old pointer is stored in R1 of the trap handler frame.

```
; R1 contains the address of the location containing
; the oldspace pointer that caused the trap
                LDEN        R2,0(R1)
; R2 contains the address of the object
; in the old space
                LDEN        R3,0(R2)
; R3 contains the first word of the object
; in the old space
; check whether the 1st word of the old object(R3)
; is forwarding or not
                CMPR        R3.TAG,FWDTAG
                BRAD        EQ,FORWARD
; If the branch was not taken,
; then we have to move the object
; Suppose here we have got the size of
; the object in R4
; and in R5 we have the address in the new space
; to which the object should be copied.
; Move the first word of the object:
                CMPR        R3.TAG,MAXIMMTAG
                BRAD        GT,POINTER1
                STNG        0(R5),R3
                BRAD        ALWS,NEXT
POINTER1:       STOG        0(R5),R3
NEXT:           SUB         R4,R4,1
                BRAD        ZE,COPYEND
                ADD         R6,R2,1
                ADD         R7,R5,1
LOOP:           LDFN        R3,0(R6)
                CMPR        R3.TAG,MAXIMMTAG
                BRAD        GT,POINTER2
                STNN        0(R7),R3
```

```
            BRAD    ALWS,LOOPEND
POINTER2:   STON    0(R7),R3
LOOPEND:    ADD     R6,R6,1
            ADD     R7,R7,1
            SUB     R4,R4,1
            BRAD    NZ,LOOP
COPYEND:    INST    R3,R5,FWDTAG
            STNN    0(R2),R3
            INST    R3,R5,R2.TAG
            STNN    0(R1),R3
            RERL
```

Some other task can write a newspace pointer into the location that holds the oldspace pointer after the trap occurs but before the trap handler starts to execute. Therefore, we have to check whether this location still has an oldspace pointer at the beginning of this program code. We can check this by executing LDET instead of LDEN at the beginning. This will cause another transport trap if the location still holds the oldspace pointer. For this modification, the trap handler needs to examine whether the trap handler itself or the user program causes the transport trap before the above program code, for example, based on the value of the trapped task's PC. Another possibility is discussed in Section 9.

12. Generation

Description The stored word is a pointer and the first operand of the address calculation has an older generation tag than the stored word.

Trap Information the effective address is saved in R1 of the trap handler

Processing After the store operation is done, the trap handler records the destination location in a table that keeps track of objects holding pointers to objects younger than themselves.

9 Conclusions and Future Extensions

MASA has a small instruction set (actually twenty-seven instructions) and a relatively large number of trap conditions (fourteen), and allows us to effectively deal with several characteristic properties of Multilisp programs. It has architectural support for tagged data manipulation, generic operations, and incremental generation scavenging garbage collection algorithms. MASA also provides special mechanisms for task activation, fast procedure invocation and trap handling, and memory-based synchronization among concurrent tasks. Since we have paid attention only to the data and control parts of the processor, the instruction fetching and memory operation units, which might implement smart memory or smart data replication features, must still be designed

The following is a list of design issues that must be decided in order to get close to a more concrete picture and of alternative design decisions that might enhance performance.

- We have not yet fixed the bit packing of instruction words. Register specifiers are versatile for tag manipulation and task management, but they take up space and we need only a few instructions exploiting this function. Therefore, restrictions might be placed on the register identifiers acceptable to most instructions. We also need to decide the size of immediate constants or branch offsets.

- With small modifications, MASA can make a conditional branch using a single instruction. The conditional branch instruction, however, has to include information needed to control both comparison and branching and thus can not afford a long bit field for the target address compared with simple branch instructions. The EQL trap can occur during a conditional branch instruction based on the EQL comparison. We have to discuss about the mechanism for returning control to the target address from the trap handler.

- CALL and ACTK instructions write two different addresses into two different tasks' PCs. One is the incremented PC to the caller task and the other is the target address to the callee task. This makes the PC

structure in the Task Database expensive, because each PC needs its own input multiplexer between the NextPC field and the Data field of the stage3 ALU register. If we perform the target address write to the callee task frame using another move instruction in advance, and CALL and ACTK only update the caller PC, then all the PCs in the Task Database can share one multiplexer and their structure gets more simple. In this case we need one more instruction for procedure call or task activation.

```
NEWCALL:    RQFR
            ADD     CHILD.R1,R2,0,MOVE
            ADD     CHILD.PC,R0,LABEL1,MOVE
            CALL    PROC1
```

- Some of trap conditions will be more conveniently enabled/disabled by STW bits instead of instruction code. For example, Transport and Generation trap conditions may only be disabled in their trap handlers. If this is true, it would be better to control enabling and disabling of these traps by STW bits and set them when a trap occurs and reset them when a trap handler terminates. We can save an instruction word bit by doing this.

- We may need to check whether a location is empty or full (or whether a location's contents are old or new) without causing a trap in some way in the Empty Location trap handler, e.g. at the beginning of the transport trap handler. Suppose a future object is implemented in two words in the memory: the value cell and the queue pointer. When it is determined, the value cell is full and contains its value or the pointer to the value object. When it is undetermined, the value cell is empty and the queue pointer points to the waiting task list. The EFbit of the queue pointer is used to exclude concurrent accesses to the shared future object during an atomic operation. When a task attempts to touch an undetermined future object, the trap handler first locks the object by setting the EFbit of the queue pointer to empty and then appends the task to the end of the waiting list. Some task may store the value and make the future object determined after the

Empty Location trap is detected but before the Empty Location trap handler starts execution. In order to deal with this race condition, we need first to make sure that the touched future object is still undetermined. The requirement for this check is the mechanism for reading the value cell without causing another Empty Location trap and examining whether it is empty or full.

- Without changing the general hardware organization, we can make it possible to overlap execution of instructions in the same task. If we overlap the first stage and the third stage executions from the same instruction stream, we have to add comparison hardware to check for the data dependencies between the instruction at the first stage and the instruction at the third stage, and extra bypassing paths from the stage3 ALU register to the stage2 register.

- Some of the trap handlers take only few instructions before returning control to the trapped task. One of the examples is the future touch trap handler as shown in Section 8. It would be possible to distinguish between long traps and short traps and to allow short trap handling to be executed in the same task frame where the trap occurred without getting a new frame allocated. To do this, some task status registers such as PC, STW, and TrapInfo need to be saved in some way.

- Since store instructions can not enable the FT or CS trap conditions, we have to check these exceptional cases by software for RPLACx operations. Store instructions might need to optionally enable these trap conditions by the strp operand.

10 Acknowledgement

This paper is written on the research work I did during my visit at MIT from July 1986 to August 1987. I joined the Parallel Processing Group (PPG) under Prof. Halstead in Laboratory for Computer Science during this period. I would like to appreciate Bert Halstead for giving me an opportunity to work with his group and advising and encouraging me through this research project. I also thank all of the PPG members for technical

discussions and valuable comments on my work. I would like to thank NEC people who are concerned with my study abroad program for supporting and understanding my activity at MIT. Finally, I thank my wife and baby (MASA) for staying with me so far from Japan.

References

[1] Baker, H.G., "List Processing in Real Time on a Serial Computer." *Communications of the ACM 21:4*, April 1978, pp.280-294.

[2] Halstead, R., "Multilisp: A Language for Concurrent Symbolic Computatoin," *ACM Transaction on Programming Languages and Systems 7:4*, October 1985, pp.501-538.

[3] Halstead, R.,et al., "The Multilisp Manual," Internal Report of Laboratory for Computer Science, June 1986.

[4] Halstead, R., "Design Requirements for Concurrent Lisp Machines," to appear in K. Hwang and D. DeGroot, eds., *Supercomputers and AI Machines*, McGraw Hill, New York, 1988.

[5] Katevenis, M.G.H., *Reduced Instruction Set Computer Architecture for VLSI,* M.I.T. Press, Cambridge, Mass., 1985

[6] Kowalik, J.S., *Parallel MIMD Computation: HEP Supercomputer and Its Applications,* M.I.T. Press, Cambridge, Mass., 1985

[7] Lieberman, H., and C. Hewitt, "A Real-Time Garbage Collector Based on the Lifetimes of Objects," *Communications of the ACM 26:6,* June 1983, pp.419-429.

[8] Moon, D.A., "Garbage Collection in a Large Lisp System," *1984 ACM Symposium on Lisp and Functional Programming,* Austin, Tex., Aug. 1984, pp.235-246.

[9] Patterson, D.A., "Reduced Instruction Set Computers," *Communications of the ACM 28:1*, January 1985, PP.8-21.

[10] Smith, B.J., "A Pipelined, Shared Resource MIMD Computer," *Proc. International Conference on Parallel Processing*, 1978

[11] Taylor, G.S., "Evaluation of the SPUR Lisp Architecture," *13th Annual Symposium on Computer Architecture*, Tokyo, Japan, June 1986, pp.414-423.

[12] Ungar, D., "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm." *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, April 1984. pp.157-167.

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 Copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 Copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 Copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 Copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 Copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 Copy
Dr. G. Hooper, USNR NAVDAC-OOH Department of the Navy Washington, DC 20374	1 Copy

END

FEB.

1988

DTIC