# SOAR ARCHITECTURE

A. Dain Samples†
Mike Klein
Pete Foley

CONTENTS



† A. Dain Samples was supported by an AT&T Bell Laboratories Scholarship.

‡ Smalltalk-80 is a Trademark of Xerox Corporation.

## LIST OF ILLUSTRATIONS

# 1. INTRODUCTION

## 1.1. The Impact of the Smalltalk-80 System on SOAR

This document is the definition of the SOAR (Smalltalk On A RISC) architecture and implementation, and is not meant to be the first exposure of a reader to SOAR, the RISC approach to architecture, or Smalltalk. This document makes several references to the Proceedings of CS292R Architectural Investigations Class, April 1983. The Proceedings' first chapter was written by Mike Klein and Pete Foley and and was the starting point for this document. Readers who are unfamiliar with Smalltalk and the RISC I architecture should not expect to understand everything in this document.

The design and environment of Smalltalk-80 has suggested certain basic directions for the design of the SOAR (Smalltalk On A RISC) architecture. Smalltalk-80 is an object-oriented system, which references objects through pointers (called OOPs, or Object Oriented Pointers). The single exception is *small integers*, which are represented by their actual integer value. *(To give you some idea of the SOAR vision, 'small' is less than 1,073,741,824.)* OOPs point to many different kinds of objects, and SOAR uses a tagged architecture to identify a few important cases. The OOP and its tag are contained in the same word, making it easy for the hardware to distinguish between different kinds of objects. To efficiently support generation scavenging, this tag also keeps track of the 'age' of an object [Unga83b].

Exception conditions (accessing an inappropriate object type, accessing objects of a certain 'age') are detected by examining the operands' tags. If a tag mismatch occurs, SOAR jumps to a trap routine to take care of the offending condition. Shadow registers are included to catch the operands and opcode that caused the trap [Blau83].

The typical Smalltalk-80 application uses deeply-nested sends, but few variables are passed and few are needed as temporaries. Thus the overlapping register window scheme is retained from RISC II, but the number of registers per window is reduced from 16 to 8. Eight register windows can buffer seven methods on the chip, covering 94% of all sends, and six registers per window are enough for 97% of all sends [Blak83]. The registers in the window are still numbered from 0 to 31 as in RISC II, but the nomenclature and structure have changed.

Smalltalk-80 objects are divided into many different classes. The method corresponding to a message could be one of the methods in the receiver's class; however, it could also be one of the methods in any of the receiver's superclasses. If the method is not one of the receiver's methods, Smalltalk-80 climbs up the superclass chain until it finds the needed method. This class searching can consume a large amount of time if it is done on every send. On most sends, however, the receiver is from the same class as the receiver of the previous message, and therefore the class where the method can be found is the same from one send to the next. To save time, SOAR supports in-line cacheing which allows the interpreter to skip over the method search code and allows SOAR to immediately execute the code for the appropriate method. Our measurements show that this optimization works 95% of the time [DAmb83].

## 1.2. Major Points of the SOAR Architecture

SOAR has full non-multiplexed 32 bit address and data busses. Addresses are 28 bits wide and the 4 MSBs of a word are tag bits, usually ignored by external circuitry. Memory words are 32 bits.

SOAR is a three-stage pipelined architecture. For the most part, this does not enter into consideration for the system programmer, but the result of some instructions can only be

understood by reference to the pipeline. These will be pointed out in the descriptions of the individual instructions where necessary. See §7 for more details on the implementation of the architecture.

SOAR is not a byte-oriented machine, and does not support arbitrary shifts (eliminating the need for a barrel shifter). Byte-oriented operations are possible, to provide compatibility with other software environments, by inserting or extracting a byte from a full 32-bit word and by treating the word as an untagged 32-bit integer. There is no need for the hardware to align bytes on loads or stores.

Upon a trap, SOAR will jump to an address determined by the opcode of the faulting instruction and the condition for the trap (vectored trapping). This allows a flexible method of handling many types of exceptional conditions. See §6.

Support for non-tagged operation is provided. Instruction formats include a bit that allow the operation to ignore the tags of the operands. This makes it possible for SOAR to support languages other than Smalltalk-80, such as C or Pascal. Smalltalk compiler writers have also found it useful when manipulating pointers [Citr83] [Laru83]. In addition, a software interrupt bit enables a special class of Smalltalk-80 interrupts for calls and jumps.

Addresses can refer to registers or memory locations. Since the registers have memory addresses, the same data is accessed whether it resides in the register file or in main memory. The appropriate bits of the memory address are compared to the Saved Window Pointer (SWP) to determine if the memory address corresponds to the address of a register (see §5).

## 2. SOAR REGISTERS

### 2.1. Description of SOAR Register Windows

SOAR register windows are similar to RISC I register windows, with a few significant changes. SOAR register windows are smaller than RISC I (eight *vs.* sixteen registers per window). Figure 1 gives the programmer's view of the SOAR registers.

| REGISTER GROUP | REG NUM | CONTENTS |
|---|---|---|
| GLOBAL | r31 | Scratch |
| | r30 | Scratch |
| | r29 | Scratch |
| | r28 | Scratch |
| | r27 | Scratch |
| | r26 | Scratch |
| | r25 | Scratch |
| | r24 | Scratch |
| SPECIAL | r23 | PSW–Program Status Word |
| | r22 | CWP–Current Window Pointer |
| | r21 | TB–Trap Base register |
| | r20 | SWP–Saved Window Pointer |
| | r19 | SHA–Shadow register A |
| | r18 | SHB–Shadow register B |
| | r17 | PC–Program Counter |
| | r16 | RZERO–Always 0 |
| HIGH | r15 | return address for this method |
| | r14 | receiver/return value |
| | r13 | arg1/local |
| | r12 | arg2/local |
| | r11 | arg3/local |
| | r10 | arg4/local |
| | r9 | arg5/local |
| | r8 | arg6/local |
| LOW | r7 | return address for called methods and traps |
| | r6 | receiver/return value |
| | r5 | arg1 |
| | r4 | arg2 |
| | r3 | arg3 |
| | r2 | arg4 |
| | r1 | arg5 |
| | r0 | arg6 |

*Figure 1 SOAR Register Window*

A word or two is in order here to explain the influence of Smalltalk on the register design of SOAR. Preliminary studies indicated that over 95% of all Smalltalk contexts do not need more than eight registers in a SOAR window [Blak83]. Smalltalk procedures (or methods as they are called in the Smalltalk jargon) use few local or temporary variables. The basic actions of a Smalltalk method on SOAR is to compute and load the parameters for the next send to an object, and do the send (which translates to a call on SOAR). The method's arguments and temporaries are in the HIGH registers. It constructs the arguments for the object it is about to call in the LOW registers. When the actual call instruction is executed, the sender's/caller's LOWs become the recipient's/callee's HIGHs. Therefore, the caller cannot use its LOWs for temporary storage across calls since any method it invokes will be using those same registers for its HIGHs.

When traps occur the system must be careful to preserve the HIGHs and LOWs of the method in which the trap occurred. If a trap handler needs its own windows/registers, then it will either need to use GLOBAL registers (very carefully!) or it will need to do the equivalent of a few call instructions to get its own window. See the discussion of window management software in §5 for some examples.

To simplify the SOAR instruction set, registers 16 through 23 have special functions (see Table 2). R16 (rzero) is the only special register available for general use (although with obvious restrictions given its nature). All others should be used only by system routines, and even then only in a highly stylized and conventionalized manner. For example, because of the pipelined architecture, data written to a special register by an instruction will not appear in the register or take effect until the second instruction following the one doing the write (i.e. special registers are not forwarded; see Appendix E). In particular, the architecture definition does not support writing into the PC register, r17.

Two other restrictions simplify the SOAR architecture. First, data may not be loaded from memory directly into a SPECIAL register; instead, the data must be loaded into a local (HIGH or LOW) register or general purpose GLOBAL register and then moved to the SPECIAL register. Data loaded into a SPECIAL register is lost, and the register remains unaffected. Second, SPECIAL registers may be used only in a particular source field (S1) and in the destination field of instructions. Using a SPECIAL register in the wrong source field yields zero for that operand. See §4.1 for descriptions of instruction formats, and Appendix E for some indication of the reasons for these restrictions.

While it is simplest to think of the special registers as 32-bit values, it should be recognized that not all 32 bits of some of the special registers exist, and not all of the existing

| Reg. | Name | Bits | *Function* |
|---|---|---|---|
| r23 | psw | 15:0 | *Process Status Word, and Destination Shadow Register (see format below).* |
| r22 | cwp | 6:4 | *Current Window Pointer. Points to one of 8 windows.* |
| r21 | tb | 31:10 | *Trap Base regieter. Location of trap vector.* |
| r20 | swp | 27:0 | *Saved Window Pointer. Points to last window saved.* |
| r19 | sha | 31:0 | *Shadow copy of A input to ALU/shifter. For interrupts.* |
| r18 | shb | 31:0 | *Shadow copy of B input to ALU/shifter. For interrupts.* |
| r17 | pc | 27:0 | *Used for PC-relative addressing & case statements.* |
| r16 | rzero | 31:0 | *Always = 0; Used to synthesize new instructions.* |

Table 2 Special Registers

bits are used. The bits that exist in each of the special registers are indicated in the *Bits* column in Table 2. The action of the SOAR chip is not defined if non-zero values are written into non-existent bits in the special registers.

Also note that the Trap Base register r21 has bits 28 through 31 even though it is an address register and therefore only bits 0 through 27 are significant. Since the architecture is defined only for a 28-bit address space, a non-zero value in bits 28 through 31 of the tb register should have no effect.

## 2.2. The PSW Special Register

PSW<15:8> contains a valid value only if interrupts are disabled. In that case it contains the opcode of the last instruction executed when interrupts were enabled. The contents of this field is undefined if read while interrupts are enabled. PSW<4:0> always contains the destination field of the last instruction executed when interrupts were enabled. If a trap occurs during the execution of an instruction, interrupts are automatically disabled thereby freezing the current values of these fields. This conveniently provides the trap handler with information regarding the instruction, and obviates the need for trap handlers to disassemble instructions.

PSW<5> is the software interrupt request bit (see §4.1). PSW<6> is the interrupt enable bit. PSW<7> is the iAPX/432 emulation mode bit, and is always undefined.

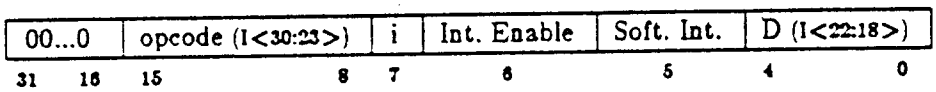


Figure 3 Format of the PSW Register

## 3. SOAR TAGS

SOAR tags are integral to each OOP (Object Oriented Pointer) and provide certain critical information about the object pointed to by the OOP. The tags are located at the beginning (MSBs) of the OOPs, and consist of either one or four bits.

Consistent with the Smalltalk-80 Virtual Machine definition, all objects are referenced by pointers with exactly one exception. To improve performance, small integers are not represented as objects. Instead, the value of the OOP, ignoring the tag, is the actual value of the small integer. Thus the architecture distinguishes a true OOP from a 'pointer' that actually contains a value. To allow the largest range of small integers, an integer tag is a single bit equal to 0. All other tags will begin with 1 and are four bits wide.

Other tags serve two general functions. The first is to keep track of the 'age' of an object for efficient support of generation scavenging [Unga83b]. SOAR objects can be divided into four different 'tenure' groups, reflecting the number of scavenging operations that the object has survived.

The other major use of the tags is to distinguish a context object's OOP from other OOPs. Contexts do not always obey a LIFO stack discipline in Smalltalk-80. Non-LIFO contexts must be specially identified, so they are not discarded in LIFO order. The context tag causes a trap on any store of a context pointer into memory so that it can be marked as non-LIFO.

The format of the tags is shown in Figure 4. Three possible tag values are undefined, and are reserved for future uses (if any).

| OBJECT POINTED TO | 32-BIT POINTER (OOP) | |
|---|---|---|
| | TAG BITS | WORD BITS |
| Small Integer | 0 | SmallInt |
| Assistant Object | 1000 | OOP |
| Associate Object | 1001 | OOP |
| Full Object | 1010 | OOP |
| Emeritus Object | 1011 | OOP |
| Context Object | 1111 | OOP |

*Figure 4 SOAR Tags*

## 4. SOAR INSTRUCTIONS

On a qualitative level, SOAR instructions are a subset of RISC I instructions. All RISC I instructions having to do with bit and byte operations have been eliminated. Single-bit shifts have been retained and byte insert and byte extract have been added. Less frequently used arithmetic instructions have also been left out. Trap instructions have been added. These instructions form a sufficient base for efficient execution of typical Smalltalk-80 programs in which little CPU time is spent performing arithmetic operations.

Instructions contain a special bit (the *tag* bit) in the opcode field that toggles tag checking. When this bit is off, SOAR executes the instruction ignoring all tags. This serves two purposes: SOAR is now compatible with programs written in standard languages such as C and Pascal, and all system support code (trap routines, garbage collector, memory allocator, and so forth) can manipulate tags as data.

There is a potential for confusion here. This document refers to the tag bit in the instruction format. When this tag bit is ONE, for example, ALU instructions trap if the operands' tags are incorrect. This is intended to be the default mode of instruction execution. When the tag bit is zero, the tags of the operands are ignored in the execution of the instruction. Since *tagged* execution is intended to be the default, the assembly language programmer does not need to do anything special. To add two operands and trap if they are not both small integers the programmer need only write:

```
add     rx,ry,rz        ;tagged add
```

but if the values in the two registers are to be treated as 32-bit integers and their tags are to be ignored, then:

```
%add    rx,ry,rz        ;UNtagged add
```

In other words, in assembly language programs, the '%' prefix to an instruction turns the tag bit OFF. Henceforth this document will refer to *tagged* (*tag* bit = 1) and *untagged* (*tag* bit = 0) instructions.

[A first time reader may wonder how the above could possibly give anyone any trouble. Historically, the tag bit was called the '%' bit, and the assembly language syntax overloaded the percent sign by using it to *turn the '%' bit OFF*. This caused a great deal of confusion over whether turning the '%'-bit ON caused tag checking (it does), or turning it OFF disabled it (it does), and with which case the percent sign should be used in the assembly language. The previous paragraph is not to educate new readers, but to re-educate old ones.]

### 4.1. Instruction Formats

Instructions follow the basic RISC I format:

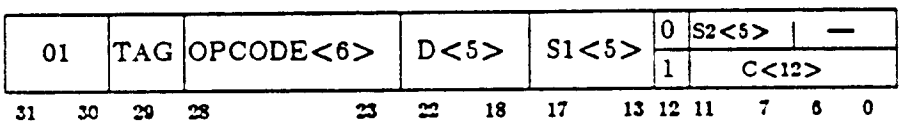


Figure 5 *Basic Instruction Format*

The D field selects one of the 32 registers as the destination of the result of the operation performed on the registers specified by S1 and S2. If bit 12 is 0, the next five bits (S2) specify another register; otherwise, the next 12 bits (C) contain a constant. Since tags occupy the upper four bits of SOAR's 32-bit data word, C specifies the upper 4 tag bits and then extends bit 7.

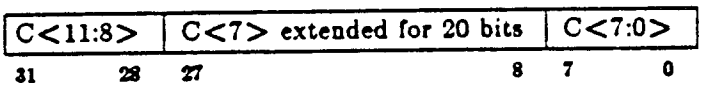


Figure 6 *Format of Immediate Constant*

Immediate constants have the above 32-bit format even for untagged instructions.

The store instruction requires a different format because it needs three operands: the base register, a constant offset, and the register containing the data to be stored. A store instruction is unlike other instructions in that it does not write a register. Stores are restricted to using only one register to form the target address.

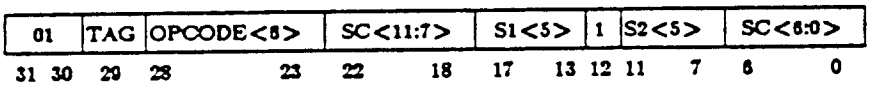


Figure 7 *Store Format*

The 12-bit store constant is broken into two fields to simplify the silicon implementation. Similar to the constant field C in the basic instruction format, the store constant SC specifies the upper four bits and then sign extends the lower eight bits. Since address computations are only 28 bits wide and tags of immediates are never checked, the tag field of SC is effectively ignored. (See §4.3 for more discussion of immediates.)

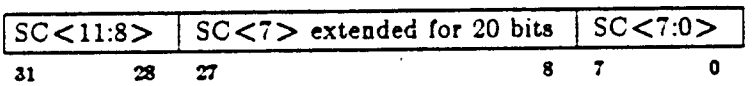


Figure 8 *Format of Store Constant*

Skip and trap instructions use the destination field of the basic format as an opcode extension to specify the condition on which the skip or trap is taken.

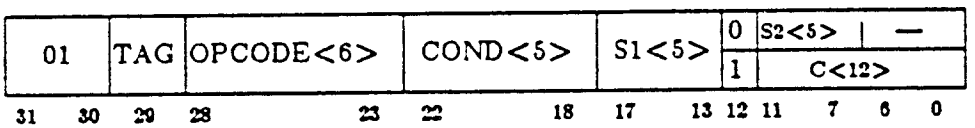


Figure 9 *Skip/Trap Instruction Format*

The call and Jump instructions use the large address format shown below:

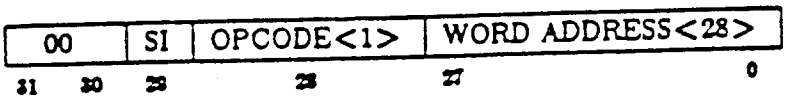


Figure 10 *Call and Jump Format*

The Software Interrupt (SI) enables software interrupts on calls and jumps if that bit is a one. This interrupt is a special class of Smalltalk interrupts that need only be checked on calls and jumps. When the system wishes to interrupt a process, care must be taken that the process is in an interruptible state. For example, if a process is interrupted in the middle of a method lookup, it is possible for the in-line cache [Samp84] [Citr83] to end up with wrong values. Therefore, the system relies on the compiler to set the SI bit in call and Jump instructions where the state of the computation is known. The system software sets the Software Interrupt Request bit in the PSW (see Figure 3) when an interrupt is desired. The hardware can then check if a software interrupt has been requested by checking PSW<5> during the execution of the call or Jump, and trap if necessary. Since the SI bit in the above instruction format occupies the same position as the tag bit in other formats, the assembly language programmer will write %call and %Jump to disable the software interrupt trap check.

### 4.2. Instruction Set

The SOAR instruction contains a subset of the RISC I instructions. The instruction set is shown in Table 11 and uses the following conventions:

| | |
|---|---|
| PC, SWP, CWP | are the registers 17, 20, and 22 respectively; |
| S1 | the number in the S1 field of the instruction; |
| S2 | the number in the S2 field of the instruction; |
| D | the number in the D field of the instruction; |
| Rs1, Rs2, Rd | the contents of registers S1, S2, and D, respectively; |
| C | the contents of the C field of non-store instructions; |
| SC | the contents of the SC field of store instructions; |
| RC | either Rs2 or C, as determined by bit 12, the immediate bit; |
| M[x] | the contents of memory location x; and |
| COND | is the contents of the conditional field of skip and trap instructions. |

Since it is intended that the default execution mode will be tagged mode, the '%' prefix means the instruction is executed in non-tagged mode. Much of the detail of instruction execution is not in the table. As long as the operands are integers, there is very little difference between the semantics of the SOAR instructions and RISC I instructions. See the following sections for the individual instructions. In these sections we will adopt the convenient notation of talking about 'instructions' and '%instructions' to distinguish tagged from non-tagged mode, respectively. The reader might like to scan §6 on traps, interrupts, and exceptions before continuing.

| I<31:30> = $01_2$ | | | | |
|---|---|---|---|---|
| Opcode | Instruction | Operands | Operation | *Comments* |
| *10-17* | [%]RET[I][N][W] | Rs1,C | PC ← Rs1+C | *Return* |
| | | | Options as part of return: | *<i>: option bit* |
| | | | [%] No trap if Rs1=OOP | *(I<29>=0) non-LIFO context* |
| | | | [I] Enable Interrupts | *(I<25>=1) PSW<1> ← 1* |
| | | | [N] R0,R1,...,R5←Nil | *(I<24>=1)* |
| | | | [W] CWP ← CWP + 1 | *(I<23>=1) Change window* |
| *50* | [%]ADD | Rs1,RC,Rd | Rd ← Rs1 + RC | *integer add* |
| *52* | [%]SUB | Rs1,RC,Rd | Rd ← Rs1 - RC | *integer subtract* |
| *44* | [%]XOR | Rs1,RC,Rd | Rd ← Rs1 xor RC | *exclusive OR* |
| *46* | [%]AND | Rs1,RC,Rd | Rd ← Rs1 & RC | *logical AND* |
| *47* | [%]OR | Rs1,RC,Rd | Rd ← Rs1 \| RC | *logical OR* |
| *40* | [%]SRL | Rs1,Rd | Rd ← Rs1 right 1 bit | *shift right logical* |
| *42* | [%]SRA | Rs1,Rd | Rd ← Rs1 right 1 bit | *shift right arithmetic* |
| *51* | [%]SLA | Rs1,Rd | Rd ← Rs1 + Rs1 | *shift left arithmetic* |
| *54* | [%]INSERT | Rs1,RC,Rd | Rd ← 0; | *insert byte; rightmost* |
| | | | byte RC<1:0> of Rd | *byte is byte 0* |
| | | | ← Rs1<7:0>. | *2 lsb specify byte* |
| *56* | [%]EXTRACT | Rs1,RC,Rd | Rd ← 0; | *extract byte* |
| | | | Rd<7:0> ← byte RC<1:0> of Rs1 | *2 lsb specify byte* |
| *94* | [%]LOAD | (Rs1)RC,Rd | Rd ← M[Rs1 + RC] | |
| *95* | LOADC | (Rs1)RC,Rd | Rd ← M[Rs1 + RC] | *For loading class of SmallInt* |
| *96* | %LOADM | (Rs1)RC,Rd | see §4.2.3 | *Load Multiple: $0 \leq d \leq 7$* |
| *90* | [%]STORE | Rs2,(Rs1)SC | M[Rs1 + SC] ← Rs2 | |
| *92* | %STOREM | Rs2,(Rs1)SC | see §4.2.3 | *Store Multiple: $0 \leq S2 \leq 7$* |
| *20* | [%]SKIP | COND Rs1,RC | if COND(Rs1,RC) PC ← PC+2 | *skip on condition* |
| *21-27* | [%]TRAPi | COND Rs1,RC | if COND(Rs1,RC) R7 ← PC, | *trap on condition* |
| | | | PC ← Trap Vector Address | *(7 different traps)* |
| *04* | NOP | | | *do nothing* |
| *05* | (internal TRAP) | | see §4.2.8 | |
| *06* | (internal SKIP) | | see §4.2.8 | |
| *60-67* | (internal LOADi) | | see §4.2.8 | *$0 \leq i \leq 7$* |
| *70-77* | (internal STOREi) | | see §4.2.8 | *$0 \leq i \leq 7$* |
| | | I<31:30> = $00_2$ (Fast Shuffle instructions) | | |
| Opcode | Instruction | Operands | Operation | *Comments* |
| *00-37* | [%]CALL | Addr | R7 ← PC, PC ← Addr | *call* |
| | | | CWP←CWP−1. | *and change window* |
| *40-77* | [%]JUMP | Addr | PC ← Addr | *jump* |

*Table 11 SOAR Instruction Set*

### 4.2.1. ALU Instructions

ALU instructions (**add**, **sub**, **xor**, **and**, and **or**) perform the expected operation on two operands. Both operands are expected to be tagged small integers and a *tag trap* occurs if they are not.

A tag trap also occurs if the result of an **add** or a **sub** instruction overflows, i.e., is greater than $2^{30}-1$ or less than $-2^{30}$. The %instructions operate on their two operands as 32-bit integers and overflow is not trapped.

As noted in section §2.1, the SPECIAL registers r16–r23 cannot be used in S2. This does not matter for the commutative ALU instructions (**add**, **xor**, **and**, and **or**). However, if you wish to subtract the contents of a SPECIAL register from any other register, the contents of the SPECIAL register must be moved to a non-SPECIAL register first.

The following table covers the operand tag traps that can occur during ALU instructions, the byte insert/extract instructions, and the shift instructions. Other conditions (e.g. ALU overflow) can also produce a tag trap and are described in the sections on the individual instructions.

| For ALU operations trap if (Rs1 is oop) \|\| (source is Rs2 && Rs2 is oop) | | | |
|---|---|---|---|
| Rs1 tag | RC tag | source | result |
| int | int | C | no trap |
| int | int | Rs2 | no trap |
| int | oop | C | no trap |
| int | oop | Rs2 | TAG TRAP |
| oop | int | C | TAG TRAP |
| oop | int | Rs2 | TAG TRAP |
| oop | oop | C | TAG TRAP |
| oop | oop | Rs2 | TAG TRAP |

Table 12 ALU Trap Logic

### 4.2.2. Shift Instructions

The shift instructions cause a tag trap if the two operands are not small integers.

The **srl** (shift right logical) instruction shifts Rs1 right one bit and inserts zeros at bit positions 30 and 31. %**srl** inserts a zero at bit position 31.

The **sra** (shift right arithmetic) instruction shifts register Rs1 right one bit with sign extension. The sign bit is bit 30 for **sra** and bit 31 for %**sra**. The difference in the sign bit is obvious. What may not be quite so obvious is the necessity for maintaining the tag bit for integer objects.

The **sla** (shift left arithmetic) instruction shifts Rs1 left one bit and inserts a zero into bit 0. A tag trap occurs if the bit shifted out of position 30 is different from the bit shifted into position 30. This instruction is redundant in that the same result could be computed with an **add** instruction. Indeed, the actual hardware implementation treats this as another opcode for **add**. The assembler simply puts the same register number in S1 and S2. However, there are instances in which it is not sufficient information simply to know that an overflow occurred on an **add**: to preserve the semantics of Smalltalk the trap handler must know that the programmer's intention was a left shift, and not an add. Hence this opcode is not superfluous or redundant. (Note that **sla** cannot have a special register as its source.)

### 4.2.3. Load/Store Instructions

The **load** instruction expects a tagged OOP in Rs1 and an integer in RC, or an integer in Rs1 and an OOP in Rs2 (immediates in the C field are always considered integers no matter what value occupies bits 28 through 31). Any other combination results in a tag trap. The address portions of Rs1 and RC are added together. The result is the address of the memory location whose contents are loaded into the specified register, Rd. Rd cannot be a SPECIAL register (r16–r23).

The **%load** instruction computes the effective address with no tag checking.

The **loadc** (load class) instruction is like **sla**: it is an opcode that can be used to pass information to a trap handler. It is a frequent operation in Smalltalk to load the class of an object. This is accomplished by loading the field of the object containing the OOP to that object's class object.

```
load        r12,ClassOffset,r13     ClassOffset is a constant
                                    and r12 contains an OOP
```

Like all objects in Smalltalk, small integers also have a class. However, trying to load the class of a small integer will result in a tag trap since both of **load**'s operands would be integers. If there were no **loadc** instruction available, the tag trap handler would have to perform a series of tests before concluding that the programmer simply was trying to load the class of a small integer which is a simple, well-defined operation. Using a different opcode for the **loadc** instruction, however, the tag trap will vector to a unique location. With programmer (or compiler) discipline that uses the **loadc** instruction for and only for loading the class of an object, the semantics of Smalltalk is maintained. The trap handler knows immediately that it was invoked because the OOP for the class of small integers was being **loadc**ed. There are no **%loadc** or [%]**storec** instructions. Table 13 summarizes the conditions which produce tag traps during load instructions.

| *On loads, trap if* (Rs1 is int && (Rs2 is int \|\| source is C)) \|\| (source is Rs2 && Rs1 is oop && Rs2 is oop ) | | | |
|---|---|---|---|
| Rs1 tag | RC tag | source | result |
| int | int | C | TAG TRAP |
| int | int | Rs2 | TAG TRAP |
| int | oop | C | TAG TRAP |
| int | oop | Rs2 | no trap |
| oop | int | C | no trap |
| oop | int | Rs2 | no trap |
| oop | oop | C | no trap |
| oop | oop | Rs2 | TAG TRAP |

Table 13 **Load** Trap Logic

The **store** instruction is analogous to **load** in both its tagged and non-tagged forms. The obvious difference is that **store** stores registers into memory and **load** loads them from memory. The not-so-obvious difference is that **store** will trap if (1) the object being stored is younger than the object into which it is being stored; or (2) the object being stored is a context. **%Store** does not check the tags of its operands or of the value being stored. In addition to the traps the **load** instructions produce, **store** instructions produce Generation

Scavenge traps as well. The store instruction traps are summarized in Table 14.

| *On stores trap if* (Rs1 is int \|\| Rs2 is context \|\| Rs2 younger than Rs1) *(Note that an integer or a context is never younger than the object it is being stored in.)* | | | | |
|---|---|---|---|---|
| (where) Rs1 | (what) Rs2 | Rs2 is younger than Rs1 | (offset) SC | result |
| int | context | ——— | int | TAG TRAP |
| int | int | ——— | int | TAG TRAP |
| int | oop | ——— | int | TAG TRAP |
| int | context | ——— | oop | TAG TRAP |
| int | int | ——— | oop | TAG TRAP |
| int | oop | ——— | oop | TAG TRAP |
| oop | context | ——— | int | GS TRAP |
| oop | int | ——— | int | no trap |
| oop | oop | False | int | no trap |
| oop | oop | True | int | GS TRAP |
| oop | context | ——— | oop | GS TRAP |
| oop | int | ——— | oop | no trap |
| oop | oop | False | oop | no trap |
| oop | oop | True | oop | GS TRAP |

Table 14 **Store** Trap Logic

To reduce the cost of overflows and context switches, SOAR has load- and store-multiple instructions that avoid multiple instruction fetches to access several operands. Primarily, they reduce the cost of window overflows and context switches, and they come only in untagged form.

The beginning address of **%loadm** is Rs1−RC, and destination field D specifies the starting register ($0 \leq D \leq 7$). The first register is loaded from M[Rs1−RC], the next register is loaded from M[Rs1−(2*RC)], and so on until R0 is loaded from M[Rs1−((D+1)*RC)]. The algorithm can be expressed as follows:

```
t ← Rs1
x ← d
Repeat
  t ← t − RC
  R[x] ← M[t]
  x ← x − 1
until x < 0
```

The beginning address of the **%storem** is Rs1−SC, and S2 specifies the starting register ($0 \leq S2 \leq 7$). The first register is stored into M[Rs1−SC], the next register is store into M[Rs1−(2*SC)], and so on until R0 is stored into M[Rs1−(S2+1)*SC]. Algorithmically:

```
t ← Rs1
x ← S2
Repeat
  t ← t − SC
  M[t] ← R[x]
  x ← x − 1
until x < 0.
```

If %loadm or %storem are interrupted (e.g. by a page fault) they must be reexecuted to guarantee completion.

%Loadm and %storem disable the pointer-to-register check on the address. Also, r0 cannot be accessed correctly the first instruction after a %loadm.

### 4.2.4. Skip/Trap Instructions

The skip and trap instructions skip the next instruction or trap to a handler, respectively, if the comparison of the operands satisfies the condition. The conditions are shown in Table 15. The condition is calculated from Rs1 − RC, and for the purposes of this table only we define Cy = 1 if there is a carry, Sn = 1 if the sign bit is a one, Ov = 1 if there is overflow, and Eq = 1 if Rs1 = RC. (There are no such program-accessible entities in the architecture.) When the instructions are executed the operands are expected to be small integers. A tag trap results if they are not. All traps that could occur for the subtract instruction described above can occur on a skip or trap instruction. When the %instructions are executed the operands are treated as 32-bit integers and no traps occur.

In addition to the signed relational conditions ($\leq$,<, $\geq$,>,=,$\neq$) and unsigned relational conditions ($\leq$,<, $\geq$,>) evaluated from the expression Rs1 − RC, SOAR also includes bounds checking. The condition INx (OUTx) where x is either 0 or 1 is true (false) if and only if $x \leq Rs1 \leq RC$. (The hardware will use the ALU to perform Rs1 − RC − 1 in unsigned mode to answer this question and assumes that RC is positive.) IN0 and OUT0 are included for use by the C or Pascal languages.

| *Result* | *Test Condition* | | *(cond, not cond)* | |
|---|---|---|---|---|
| | *Rs1 op RC* | *arith* | COND (I<=18>) | Mnemonic |
| Eq | *equal* | | $04,05_8$ | EQ,NE |
| Sn xor Ov | *less than* | *2's comp* | $02,03_8$ | LT,GE |
| (Sn xor Ov) or Eq | *less than or equal* | *2's comp* | $06,07_8$ | LE,GT |
| Cy | *less than* | *unsigned* | $12,13_8$ | LTU,GEU |
| Cy or Eq | *less than or equal* | *unsigned* | $16,17_8$ | LEU,GTU |
| 0 | *never* | | $00,01_8$ | NEVER,ALWAYS |
| Cy | $0 \leq Rs1 < RC$ | | $12,13_8$ | IN0,OUT0 |
| Rs1≠0 and (Eq or not Cy) | $1 \leq Rs1 \leq RC$ | | $22,23_8$ | IN1,OUT1 |

*Table 15 Test Conditions for* skip *and* trap *Instructions*

There are seven trap instructions (**trap1**, **trap2**, ... **trap7**) with their own unique trap vector locations that allow the programmer to define one-cycle test instructions.

### 4.2.5. Byte Instructions

SOAR uses word-oriented addressing. The instructions **Insert** and **extract** have been designed to make life easier for programs that must manipulate byte data. These instructions expect both operands to be small integers and will take a tag trap if they are not. **Insert** uses RC<1:0> to select a byte position in Rd (bytes are numbered in increasing order right to left, the rightmost byte is byte0), and the least significant byte of Rs1 is placed there. The rest of Rd is zero. **Extract** uses RC<1:0> to select a byte of Rs1 and stores it in the least significant byte position of Rd. The rest of Rd is zero.

**%Insert** and **%extract** do not require their operands to be small integers and will not cause a tag trap.

### 4.2.6. Call/Jump Instructions

The **call** instruction decrements the register window pointer, stores the return address in r15 of the new window, and branches to the target address. The return address is the address of the **call** instruction plus one. The **%call** instruction does all of this while ignoring software interrupt requests.

A **call** instruction will cause a Window-Overflow trap if, at the beginning of its execution, the following is true:

$$((CWP - 0x10)\ \&\ 0x70) == (SWP\ \&\ 0x70)$$

Note that '%' does *not* disable the check for window overflow.

The **jump** instruction simply branches to the indicated location if there is no outstanding software interrupt request. **%jump** ignores software interrupt requests.

If there is a software interrupt request outstanding when **call** or **jump** is executed, a Software Interrupt trap is taken.

### 4.2.7. Return Instruction

Built into the **ret** instruction are many of the mechanisms needed for the transition between different states of Smalltalk execution. Its basic action is to get the return address from Rs1 and branch to the return address plus RC. There are no tag traps with the **ret** instruction, but there is a Generation Scavenge trap that is taken if the value in Rs1 is a pointer and not an address (all 28-bit addresses in a 32-bit word look like small integers). See [Samp84] for a use for this mechanism. This trap is disabled for **%ret**.

**reti** enables interrupts after returning, meaning that the return will not be interrupted (assuming, of course, that interrupts were disabled during its execution).

**retn** will nil registers 0 through 5 after changing the CWP, the current window pointer. This is very important for Smalltalk's (or any system's) garbage collection where hanging pointers could wreak havoc. Also, the semantics of Smalltalk specifically state that temporary variables' initial values will be nil. It is easier to prepare the way for future calls upon return, rather than performing the nil operation during a call when a window overflow would cause a trap. Nil's value is 0xB000000.

**retw** provides the full inverse of the **call** instruction by also incrementing the register window pointer before branching to the return address.

All combinations of the above suffixes are allowed: **retin**, **retiw**, **retnw**, and **retinw**. The only affect of %instruction for any form of return is to disable the tag check on the value in Rs1. Of course, if Rs1 does not contain an address, and the trap is not taken, an error will probably result.

It is important to understand the effective order of the operations of the return instruction:

(1) read the return address from Rs1;

(2) take a Generation Scavenge trap if Rs1 is not an address (tagged mode only);

(3) take a Window Underflow trap if ((CWP + 0x10) & 0x70) == (SWP & 0x70);

(4) if 'w' then change the window pointer;

(5) if 'i' enable interrupts;

(6) if 'n' then nil registers 0 through 5 (thereby preserving the return value and the return address in R6 and R7, respectively);

(7) branch to the return address.

### 4.2.8. Internal Instructions

Most instructions go straight through the pipeline, effectively executing in one machine cycle. Some instructions (e.g. **load**, **store**, **load**, **store**, and *ret*) require more than the one machine cycle allotted. If these instructions are interrupted by a trap, it is sometimes necessary to know where the instruction was in its multi-cycle execution in order to return correctly from the handler.

A prime example is the **skip** instruction: if the skip condition is satisfied, the next instruction in the instruction stream is skipped, even though it is still fetched. If fetching that next instruction causes a page fault trap, handling that trap would lose the information as to whether the first instruction on the new page is to be executed or not.

The architecture handles this problem and others like it by defining a set of "internal" opcodes that are placed in the pipeline at appropriate points. Some of these opcodes are merely place holders, others retain information that would otherwise be lost (as in the skip example), while others have mini-functions associated with them. Only one of these opcodes is available to the user, **nop**. In all other cases, however, the opcodes were designed to be used only by the architecture in a defined sequence and not by the user. Even though the user can put these opcodes in the instruction stream himself, their execution is undefined if they are used in that manner.

More information on the exact nature of these opcodes can be found in §7. Ideally, these opcodes should be hidden from the user – i.e. an illegal opcode trap occurs if a user tries to use them – but for historical reasons this has not been implemented.

## 4.3. Immediates

The immediate fields C and SC must be used with care. Being able to construct OOPs as well as integers in the immediate field is useful, but also a potential source of programming error. Due to the pipeline, the instruction:

```
thisLoc:    add         pc,0,r0
```

leaves thisLoc+1 in r0. PC-relative addressing is accomplished with:

```
thisLoc:  %load     (pc)dataLoc – thisLoc – 1, r0
          :
dataLoc:  data
```

This is valid for C and SC values in the range 127 to -128.

The assembly language programmer using **sa** should be aware that the following code will produce a tag trap:

```
        add         rzero,-1,r0
        add         r0,-1,r1                *tag trap!*
```

The problem is not the -1 in the second **add** where the trap is taken. The problem is the '-1' in the first instruction! If the -1 immediate is stored in the C field of the first **add** as 0xfff then when the C field is expanded into a 32-bit integer, the resulting value stored in r0 will be 0xffffffff. This is not a small integer: it is a context OOP. (Notice the upper four bits.) Hence the tag trap. For this reason, **sa** (the SOAR assembler) has the '#' notation for specifying the tag field of immediates *even though the hardware does not check the tag field of immediates in any instruction.* All immediates are treated as 32-bit integers. In the immediate field, n#m ($n \neq 0$) specifies an immediate whose tag bits are n and the lower eight value bits are m. If n = 0, the specification is a little more difficult. Suffice it to say that all integers from 0#-128 to 0#127 are representable (but see Appendix C). Therefore, the correct code is:

```
        add         rzero,0#-1,r0
        add         r0,0#-1,r1              no tag trap
```

One consequence of the SOAR instruction set that may not be readily apparent is the lack of the ability to load arbitrary 32-bit constants, or to load the contents of an arbitrary location from memory with one instruction. Since the number of significant bits in the immediate field of all instructions is eight bits, offsets can only be in the range -128 to +127. (Allowable integer constants are not so simply specified: see Appendix C).

## 5. REGISTER WINDOW MANAGEMENT

Register windows forced out of the internal register file by an overflow condition are saved into a register overflow area in memory pointed at by the Saved Window Pointer (SWP). The first window saved into this area is saved at the highest address location and additional windows are added at lower address locations. (See Figure 16)

The SWP indicates the division between the saved register space and the memory space that is currently mapped into the register file. SWP<6:4> contains the 'window number' of the next window that would be reloaded from memory on window underflow. Or, to state the dual, SWP<6:4> contains the number of the last window spilled to memory because of window overflow. The SWP thereby acts as a separator between those windows that have been spilled to memory and those still in the register file.

To maintain consistency and simplify system software, pointers can access a context's registers whether they are on the chip or have been spilled into memory. If a 28-bit address A<27:0> satisfies the following two tests then it to refers to an on-chip register:

(1) *A<3> = 1.* As mentioned above, all context objects have headers containing the object's class, length, and so forth. Thus, if A refers to locations 0 though 7 of a context object, the reference is to the header in memory. However, if A refers to locations 8 through 15, the reference is to the context's registers.

(2) *(SWP<27:4> − A<27:4> − 1)<27:7> = 0.* This test determines if the memory access refers to any of the eight register windows on the chip. The action of the chip is undefined if the SWP allows the overflow area to 'wrap-around' memory, i.e. SWP < 0x80.

An overflow condition occurs (and a trap taken) if a **call** instruction is about to be executed and (CWP−SWP)<6:4> = 1. An underflow condition occurs (and a trap taken) if a return instruction is about to be executed and (SWP−CWP)<6:4> = 1.

SOAR has 8 register windows in its internal register file. The overflowed windows are mapped to the memory block whose address bits <6:4> are the same as the window number.

When overflow occurs, one window will be swapped out to the register spill area. Since everything is an object in Smalltalk-80, including contexts, this window will be provided with an object header when it resides in memory. The header needs some space (four words at the present) in addition to the eight words for the registers. To simplify the hardware, each saved window will take up sixteen words. The logical and physical representations of a context object at location *obj* are shown in Figure 17. Also shown are suggestions for Smalltalk's use of the rest of the object.

To create a pointer to a context (i.e. the location of the context object using these registers) calculate the address of the current window. Using 'rt' to be the result register, the SOAR code is:

```
%add    cwp,0,rt        move to non-special register
%sub    swp,rt,rt       compute distance of this window from
                        the last-saved window
%and    rt,0x70,rt      isolate the window number
%sub    swp,rt,rt       compute the address
```

To turn this into a context pointer (OOP), set the upper 4 bits of rt to all ones (i.e. 'or' in the context tag 0xf0000000). Appendix A provides more explanation of why the above code works.

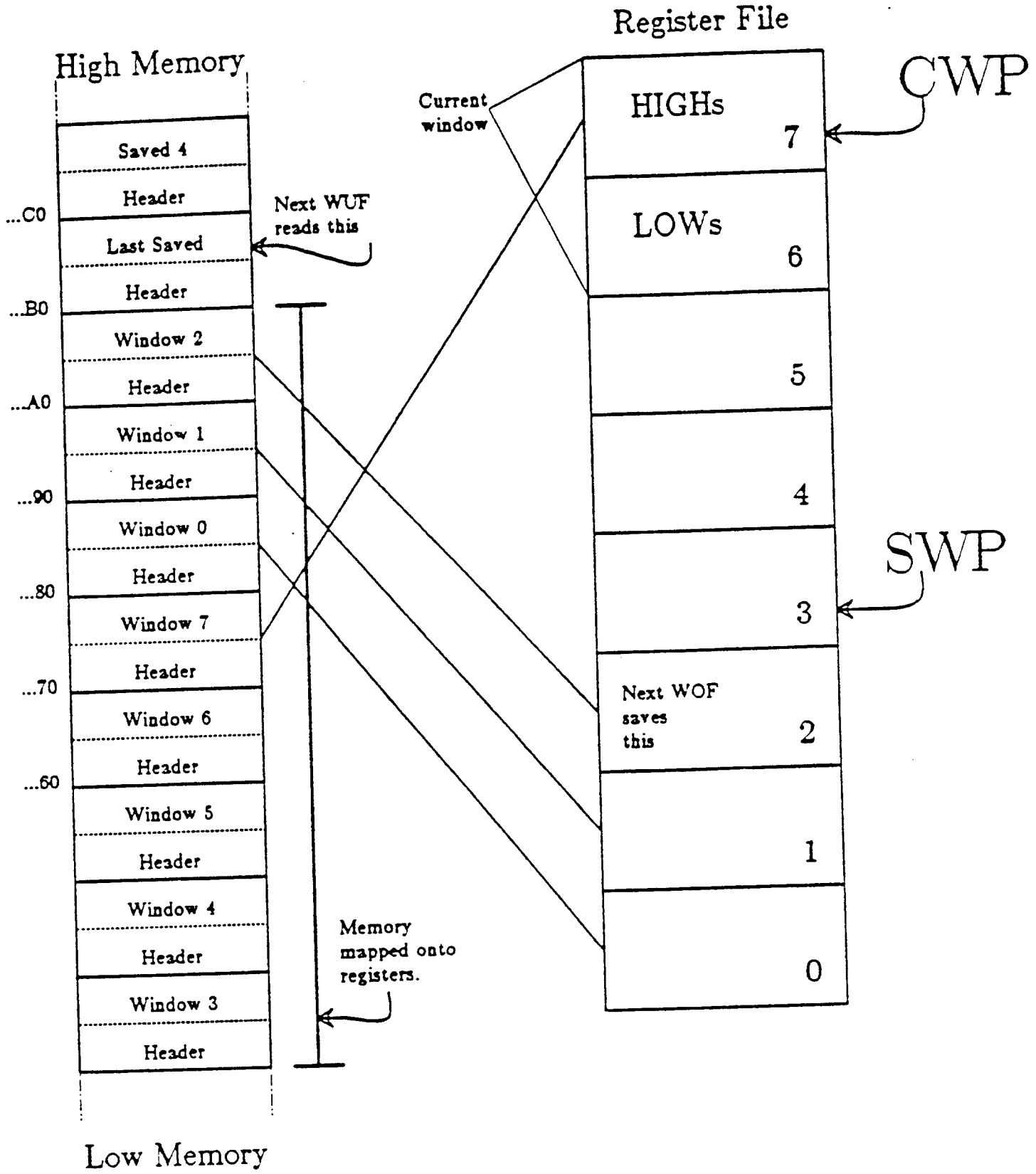


*Figure 16 SOAR Saved Register Space*

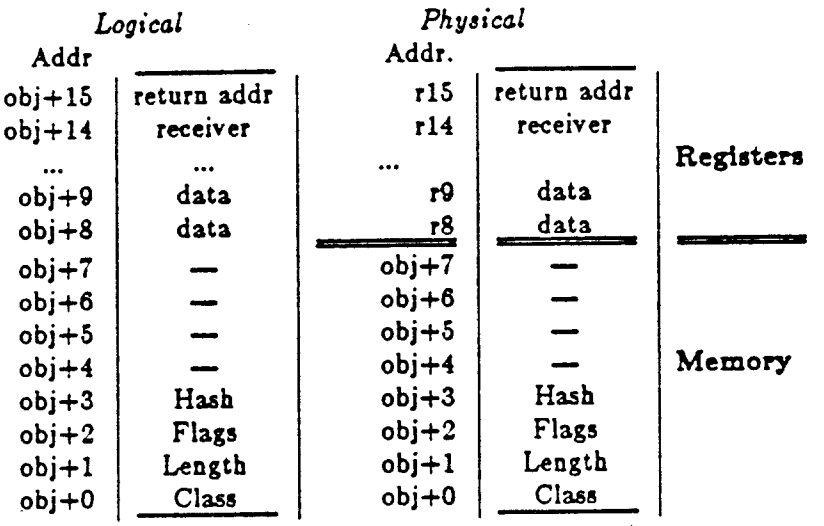


*Figure 17 Logical and Physical representations of Contexts*

The code to test if the execution of a **call** instruction would produce a window overflow trap (rt is a temporary register):

```
%add    swp,0x10,rt     get swp + 1
%and    rt,0x70,rt      get the number of the next window
                        to be saved
%skip   ne cwp,rt       you're OK if they're not equal
%jump   FULLSTACK       otherwise, handle a full register file
```

To handle a window overflow trap during the execution of a **call**: (the return address is in r7 on entry, and rt must be a GLOBAL register)

```
%sub    cwp,0x10,cwp    move the window to be saved into
                        the LOWs; also "moves" the return
                        address into r15
%add    swp, 0, rt      the actual address+1 of where
                        the registers will be stored
%sub    swp,0x10,swp    compute the context OOP;
                        the window we want to save is
                        in the LOWs
%storem r7,(rt)1        store registers seven through zero
                        into M[rt-1] through M[rt-8]
%retiw  r15,-1          return to re-execute the call
```

Handling a window underflow on a return instruction is a little more subtle:

```
%add    cwp,0x20,cwp    position the window to loaded in the LOWs
%add    swp,0x10,rt     compute the actual address+1 of the registers (also lets cwp settle)
%loadm  (rt)1,r7        load registers seven through zero
%sub    cwp,0x20,cwp    reset the windows
%add    swp,0x10,swp    point at the next window to be reloaded
%reti   r7,-1           re-execute the faulting return instruction
```

See Appendix B for a specific example of how spilling register windows works.

# 6. TRAPS AND INTERRUPTS

SOAR employs a flexible trapping philosophy to replace the usual execute-on-condition instructions and to provide for more hardware support for Smalltalk. For example, trapping on tags allows SOAR to update data structures for dynamic memory management very efficiently. The cost of the various checks required by some memory management mechanisms is absorbed in the tag-checking of individual instructions. This section quotes from [Unga83a].

## 6.1. Terminology: Interrupt and Trap Condition Names

Table 19 lists the trap conditions, the value of the four VECTOR bits for that trap type, and groups the traps in order of priority. The trap vector is formed from the TB register, the VECTOR bits for the trap type, and the opcode of the instruction that was executing when the trap condition was detected. This is the opcode of the instruction causing the trap except for instruction page faults. For instruction page faults it is the opcode of the previous instruction (i.e. the last instruction on the previous page, or the instruction that jumped into the missing page).


Figure 18 *Trap Address Format*

The trap handler can obtain the opcode of the trapping instruction from bits <15:8> of the SHDST/PSW register as well as from the trap address.

This document uses the terms 'traps' and 'interrupts' as if they are somehow different. In fact, software interrupts, hardware interrupts, and exceptional conditions all use the same trapping mechanism. Despite this, our vocabulary tends to follow traditional usage in which the word 'trap' refers to an exceptional condition caused by the executing instruction.

| Name | Vector | Pri Class | Explanation |
|---|---|---|---|
| ILL | 0 | A | illegal opcode: I<31>=1 *or* (I<31>=0 *and* I<28:23>=unused) |
| TT | 1 | B | tag trap: illegal tags or ALU overflow |
| SWI | 2 | B | software interrupt |
| WO | 3 | C | window overflow (calls) |
| WU | 4 | C | window underflow (on returns) |
| DPF | 5 | C | data page fault |
| TI | 6 | C | trap instruction |
| GS | 7 | D | GS trap(store new into old, store context, nonLIFO ret) |
| IPF | 8 | E | instruction page fault |
| I/O | 9 | F | I/O request |

*Table 19 Interrupts in Priority Order (A is highest)*

'Interrupt' refers to an exceptional condition raised external to the executing instruction.

In order to make SOAR an efficient Smalltalk vehicle, it has been designed so that programs can leave crucial information in the instruction stream. The primary examples are the functionally redundant **loadc** and **sla** instructions. Not so obvious is that *all* 64 opcodes in I<28:23> are available to the programmer. Because of the trapping mechanism, illegal opcodes become software definable "extracodes" to "expand" the instruction set: there is a unique trap vector for each illegal opcode. Those opcodes that are not recognized by SOAR will take an illegal opcode trap. The programmer can then write a trap handler that will execute a user-defined function. An obvious and simple application is the setting of different flavors of breakpoints in a debugger. The primary advantage of using illegal opcodes in this fashion (as opposed to just inserting calls in-line) is that data can be stored in the rest of the instruction either by the compiler or the program. The trap handler can easily retrieve the data. (Note, however, that an illegal opcode trap precludes meaningful data in any but the opcode shadow register.)

The **trap** instructions have been designed to give the user flexibility in the design of single cycle test instructions. Since special action (a trap) is taken only if the condition is satisfied, the program is penalized only one cycle for the test. Obviously this can be used to advantage if the condition being tested must be checked frequently, has a short code sequence to compute the condition, and has a global action to be performed if the condition is true.

### 6.2. Trap Priorities

Although there may be more than one reason for a trap to occur, the hardware will select only one of them. To simplify the interface to the trap handler code, the reasons are prioritized. The lowest priority traps are *I/O* and *Instruction Page Fault* because these need not be serviced immediately. If *I/O* or *IPF* occur simultaneously with another trap, the other trap is handled, and the *I/O* or *IPF* occurs when returning from the interrupt handler. The general rule in the case where there are two traps that must be serviced simultaneously is that the less frequent one is given higher priority and its trap handler must check for the more frequent trap. Generally, this is most simply done by re-executing the faulting instruction, if possible. Otherwise, the other trap possibilities must be checked and, if a fault condition exists, a trap call emulated.

Thus the more frequent traps are faster because they need not take the time to check for the other interrupt conditions. For example, a return instruction may incur a tag trap (non-address in the return address register), and it may cause a window underflow at the same time. The tag trap is given higher priority and handled first. Before returning, the trap handler must also check for window underflow. There are two things to note. First, window underflow is the only trap condition that must be checked in this particular example. None of the other traps are possibilities. Second, in this particular example the easiest way to check for window underflow is to put a legal return address in the return register and reexecute the return instruction that caused the initial fault.

Table 20 lists, for each instruction family, the possible traps and interrupts from highest to lowest priority that can occur for each class of instruction.

### 6.3. Capturing Information and Reexecuting Instructions

The shadow registers in SOAR capture all the information necessary to allow any interrupted instruction to complete. The shadow registers are loaded each cycle when interrupts are enabled. When a trap is taken interrupts are automatically disabled, thereby preserving the operands of the faulting instruction. Trap handlers must use only those

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Call | ILL | SWI | WO | | IPF | I/O |
| Jump | ILL | SWI | | | IPF | I/O |
| Return | ILL | | WU | GS | IPF | I/O |
| ALU | ILL | TT | | | IPF | I/O |
| Skip | ILL | TT | | | IPF | I/O |
| Trap | ILL | TT | TI | | IPF | I/O |
| Shift | ILL | TT | | | IPF | I/O |
| Load | ILL | TT | DPF | | IPF | I/O |
| Store | ILL | TT | DPF | GS | IPF | I/O |
| Byte | ILL | | | | IPF | I/O |
| Nop | ILL | | | | IPF | I/O |
| (STOREi) | * | | DPF | | * | I/O |
| (LOADi) | * | | DPF | | * | I/O |
| (SKIP) | * | | | | IPF | I/O |
| (TRAP) | * | | | | * | I/O |

(* can occur only if the instruction is in the instruction stream)

*Table 20 Interrupt Priority per Instruction Family (left-to-right)*

instructions that cannot produce traps (including page faults!) until the shadow registers are saved. This means that the handler cannot use tagged instruction, calls, rets, or loads or stores that could cause data page faults. Instruction page faults are prevented by keeping the handler pages locked in memory. If a trap handler (mistakenly) causes a trap before saving the shadow registers, the trap address would be computed *using the previously shadowed opcode.* The newly invoked trap handler would almost certainly be handling the wrong trap with the wrong operands in the shadow registers.

However, safe use of non-tagged instructions will store the operands and enable interrupts (which enables operand shadowing). At that point, all instructions are again available for use.

SOAR saves the PC in R7 on a trap or interrupt. The value in that register after a trap is the address plus one of the instruction whose execution was interrupted. The first instruction at the trap address must be a **jump**.

### 6.3.1. An Apparent Ambiguity Explained

Because the trap address is formed from the trap vector and bits <28:23> of the instruction, and because use those bits of the instruction as part of their address field, it is possible for a trap occurring during a **call** instruction to vector to one of many possible locations depending on the upper bits of the address field. Furthermore, it is possible for traps occurring during **call** and **jump** instructions to vector to locations used by other opcode/trap vector pairs. However, SOAR has been designed such that this causes no ambiguity. The only overlapping vectors are for the Instruction Page Fault and the I/O Interrupt, and neither of these care which instruction's opcode is used to invoke the handler.

# 7. SOAR IMPLEMENTATION

## 7.1. Major Points of the Implementation

SOAR is a three-stage pipelined architecture. The standard SOAR instruction requires three machine cycles to traverse the pipeline: InstrFetch, Operate, WriteResult. Each machine cycle is composed of three non overlapping clock phases, PHI1, PHI2, and PHI3 (see Table 23 and Figure 24). Some instructions (**load**, **loadm**, **loadc**, **store**, **storem**, **trap**, and **ret**) require extra cycles. For example the **load** and **store** instructions require four cycles: InstrFetch, Operate, Ofetch/Ostore, and WriteResult. Because of the three-stage pipeline, a full instruction is effectively executed in each machine cycle except for the noted instructions. Table 21 shows results from recent simulations that indicate that SOAR should operate with a 500ns machine cycle.

The SOAR design incorporates a concept called the **Fast Shuffle**™. Since the processor is fetching the next instruction while executing the current instruction, it is necessary to detect those conditions when the next instruction to be fetched is not in the next sequential location in memory.

All latches internal to SOAR are pseudo-static, so that debugging and testing can proceed in more flexible ways, and so that external devices can force SOAR to wait for arbitrary lengths of time, as might be expected when SOAR is used with another processor [Blom83].

## 7.2. SOAR Block Diagram

Figure 22 is a block diagram of the SOAR architecture.

## 7.3. The Pipeline

When counting the number of memory cycles a piece of code will require to execute, the following rules obtain:

(1) **Load** and **store** instructions take two cycles.

(2) **Storem** and **loadm** take the number of cycles equal to the number of registers being stored/loaded, plus one.

(3) **Ret** instructions take two cycles.

(4) **Skip** instructions take one memory cycle. If the skip condition is satisfied and the following instruction is skipped, add one cycle; if the condition is not satisfied, then

| | |
|---|---|
| Pre Charge | 150 ns |
| Register Read | 130 ns |
| ALU Operation | 160 ns |
| Non Overlap | 60 ns |
| TOTAL | 500 ns |

*Table 21 Predicted Cycle Times for SOAR*

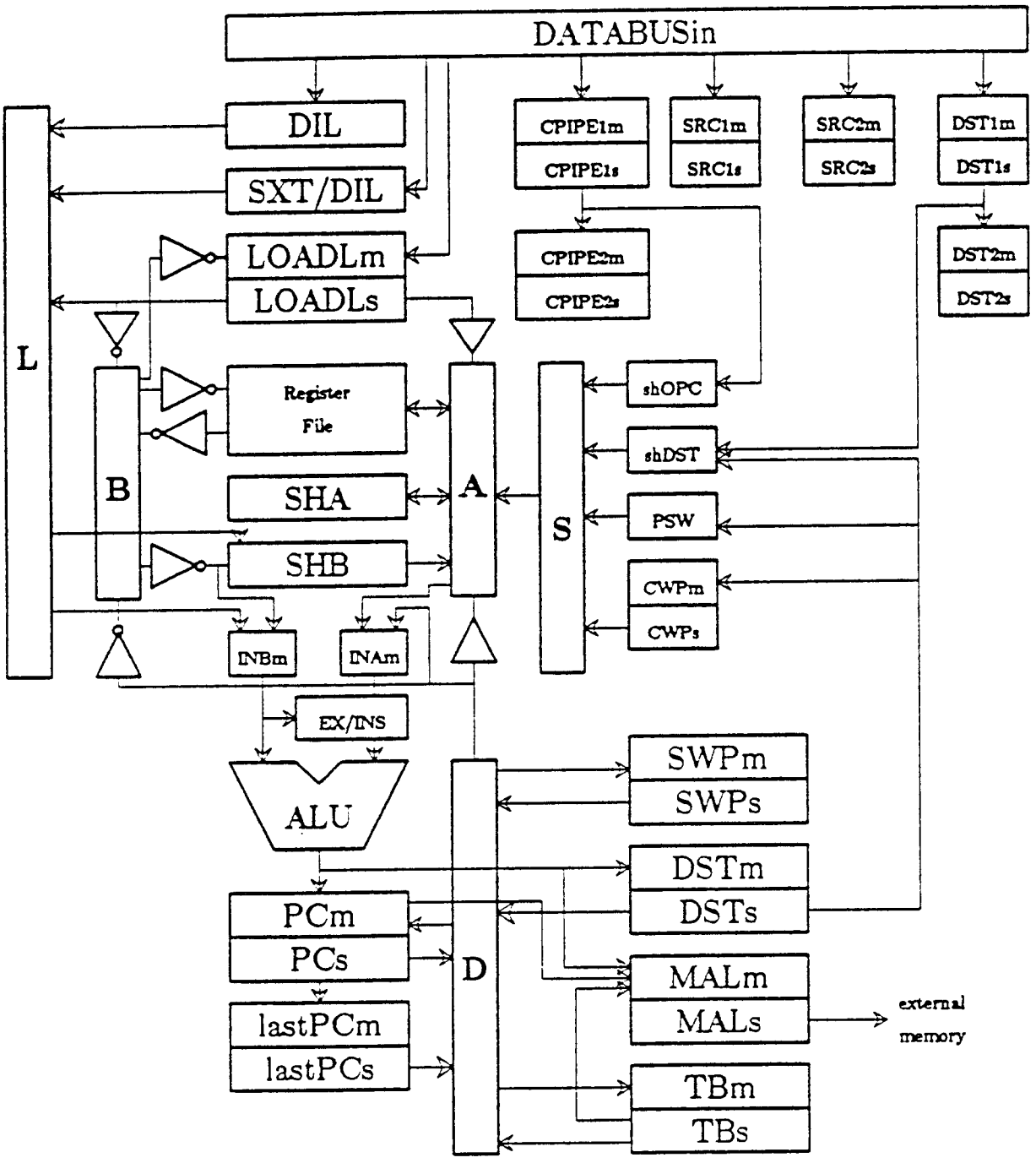


*Figure 22 SOAR Block Diagram*

count the following instruction normally.

(5) **Trap** instructions take one memory cycle if the trap condition is *not* satisfied. Add one cycle if the condition is satisfied and the trap is taken. (Don't forget to count the **Jump** instruction in the trap vector.)

(6) All other instructions take one memory cycle.

How this is achieved with the pipeline architecture is described in the following sections.

### 7.3.1. Standard Instruction Execution

The majority of the SOAR instructions execute in three cycles. Because of the pipelined architecture, these cycles overlap with the cycles of other executing instructions. Each cycle consists of three phases that are described in Table 23. Figure 24 depicts how the cycles of executing instructions overlap. In the following sections, we will often refer to the instructions in that figure as I1, I2, I3, ... That is, instruction I1 is the instruction fetched during IF1, etc.

Referring to Figures 15 and 17, we can trace the flow of standard instructions through the SOAR pipeline. Assume that we are reading the first instruction onto the chip from location *Addr*.

| | | |
|---|---|---|
| Cycle 1 | PHI1 | Instruction Fetch |
| | PHI2 | Instruction Fetch |
| | PHI3 | Instruction Fetch |
| Cycle 2 | PHI1 | Pre Charge |
| | PHI2 | Register Read |
| | PHI3 | ALU Operation |
| Cycle 3 | PHI1 | |
| | PHI2 | |
| | PHI3 | Register Write |

*Table 23 SOAR Instruction Cycle*

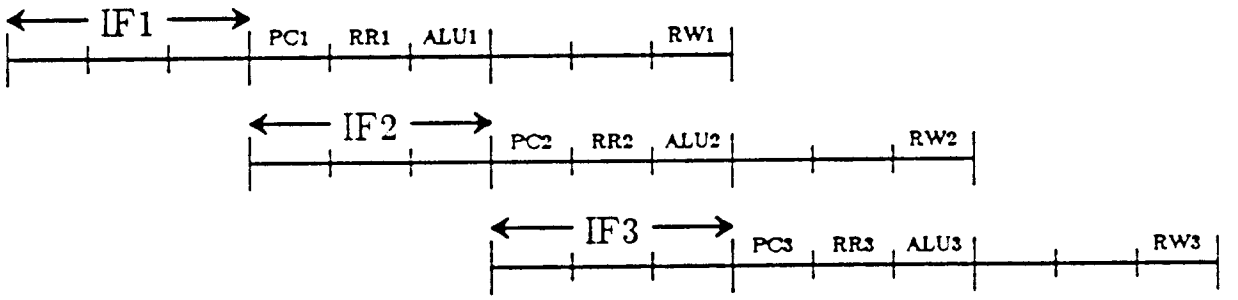


*Figure 24 Timing for Standard Arithmetic/Shift Instruction.*

**Cycle 1**: The address *Addr* of the instruction is sent to the external memory circuitry in PHI1 of IF1. By PHI3 of IF1, the content of that memory location is available on the DATABUSin and is loaded into CPIPE1m, SRC1m, SRC2m, and DST1m which correspond to the opcode, S1, S2, and D fields of the instruction. We will call this instruction I1.

**Cycle 2**: At the beginning of the next cycle, the address *Addr*+1 of the next instruction (I2) goes out to external memory, and instruction I1 is clocked into CPIPE1s, SRC1s, SRC2s, and DST1s. The opcode in CPIPE1s controls the register read during PHI2 and the ALU operation during PHI3 for instruction I1. The result of the ALU operation is placed in the DSTm. Also in PHI3, I1 is sent to CPIPE2m and DST2m, and I2 is sent to CPIPE1m, SRC1m, SRC2m, ans DST1m.

**Cycle 3**: The address *Addr*+2 of I3 is sent to the external memory, I1 is clocked into CPIPE2s and DST2s, and I2 is clocked into CPIPE1s, SRC1s, SRC2s, and DST1s. Writing the result of I1 (sitting in DSTs at this point) is controlled by the opcode in CPIPE2s. The register operands for I2 are read during PHI2, and the result of I1 is written into the designated register during PHI3.

As you can see, even in this simplified picture of how instructions find their way through the pipeline, there are many overlapping tasks being performed by the hardware. The overlapped nature of the execution cycle imposes some restrictions on the programmer that may not be understandable from a traditional fetch/execute view of the instruction cycle. Modifications to the above paradigm and explanations of non-obvious consequences are in the following sections. Appendix E contains an encapsulation of these restrictions.

As a shorthand, any movement of data from, e.g. CPIPE1m, SRC1m, SRC2m, and DST1m into CPIPE1s, SRC1s, SRC2s and DST1s will be indicated by just noting that CPIPE1m is moved into CPIPE1s, with the movement of the rest of the decoded instruction implicit. Also, it should be noted that a common technique for handling special situations is to 'force' the internal opcodes into the pipeline, usually into CPIPE1s.

### 7.3.2. Register Forwarding

As can be seen from Figure 24, if I1 stores its results in, say, r9 and I2 is to read r9 as one of its operands, we have a problem. I1 does not store its result in r9 until PHI3 of its third cycle (RW1), well after I2 reads r9 in PHI2 of its second cycle (RR2). Instruction decode detects this conflict and 'forwards' the result of I1 to the register read phase of I2; i.e. the value for r9 is not read from the register file, but is obtained from DSTs. However, this is done only for registers 0 through 15 (LOWs and HIGHs) and registers 24 through 31 (GLOBALS). The SPECIAL registers are not forwarded.

### 7.3.3. Fast Shuffle Instructions and Return

The **call** and **jump** instructions produce another kind of perturbation in the nice three-cycle execution model described above. In Figure 24 we see that, if the instruction fetched during IF2 is a **call** or **jump** and this is not acted on until its second cycle, then the next instruction in the instruction stream will already be in the process of being fetched (IF3). Therefore, the chip checks *during instruction fetch* to see if the incoming instruction is a **call** or **jump** (actually the check is made when the instruction is initially loaded into CPIPE1m during PHI3 of the instruction fetch). If the instruction is a **jump** or **call**, the hardware does a Fast Shuffle: the target of the **call** or **jump** is used as the address at which the next instruction is to be fetched. The instruction fetched during IF3 is then the correct instruction and no cycles are wasted. See Figure 25.

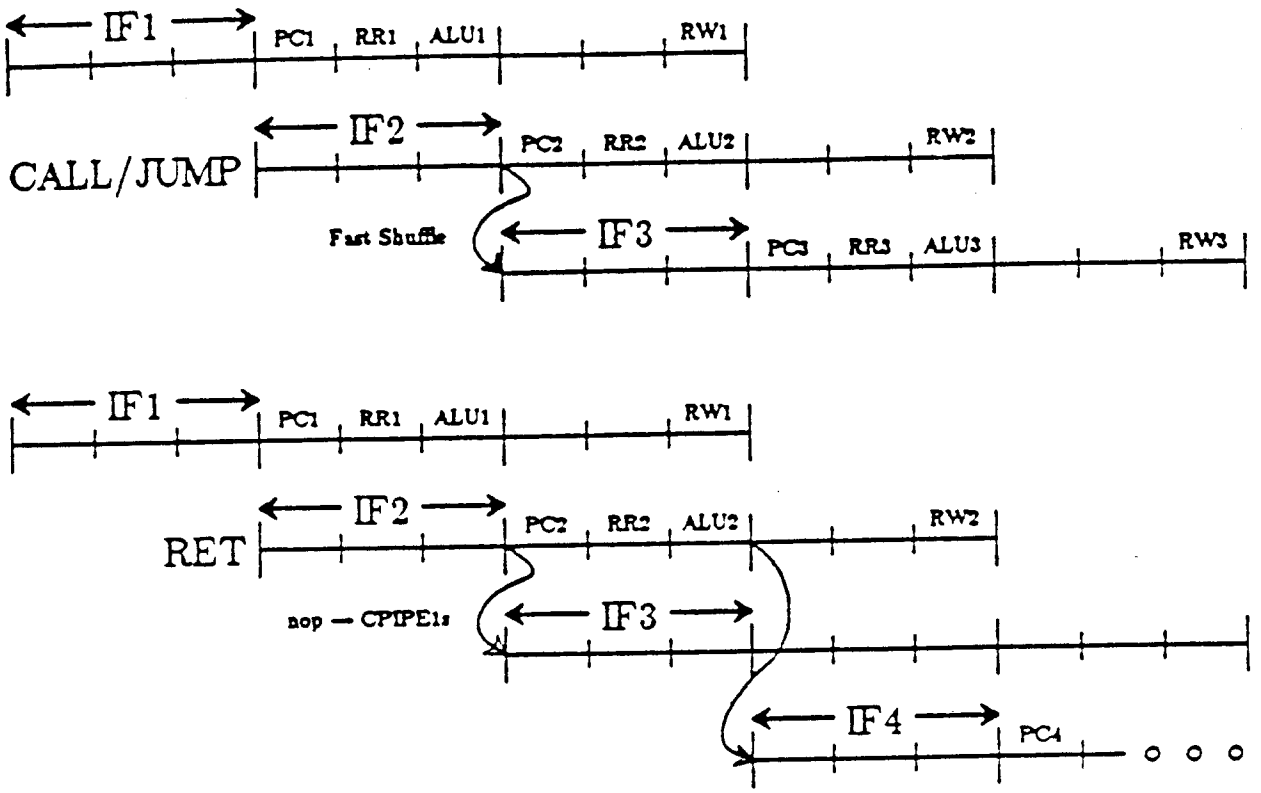


*Figure 25 Timing for* **call/jump** *and* **ret**

If I1 is a **skip** instruction, and the just-fetched I2 is a **call** or **jump**, then the Fast Shuffle mechanism will be used only when the skip condition is not satisfied. If the condition is satisfied then the following instruction's opcode is replaced with the internal instruction SKIP.

The **ret** instruction also faces a similar problem as the **call** and **jump**, with the added complication that its target is not known until the end of its second cycle. (Therefore the Fast Shuffle mechanism cannot be used to prevent the fetch of the next instruction in the instruction stream.) IF I2 is a **ret**, the instruction following it in memory never finds its way into CPIPE1s: the **ret** instead forces a **nop** opcode into CPIPE1s and **nop** becomes I3. The result of the address computation during the **ret**'s ALU operation is fed directly into PC and the MAL in PHI3 of the **ret**'s second cycle. I4 is the first instruction at the return address.

### 7.3.4. Load and Store Instructions

The **load** and **store** instructions require two memory accesses to execute: instruction fetch and operand fetch/write. The target address of the **load** or **store** is computed in the instruction's second cycle and is therefore not available until the instruction's third cycle. Also, **storeing** or **loading** during the third cycle conflicts with the instruction fetch of the next instruction from memory (see Figure 26).

Let us work through how a **load** instruction works. During the second cycle of I2 the instruction following the **load** in memory is fetched. By the end of PHI3 of IF3, the new

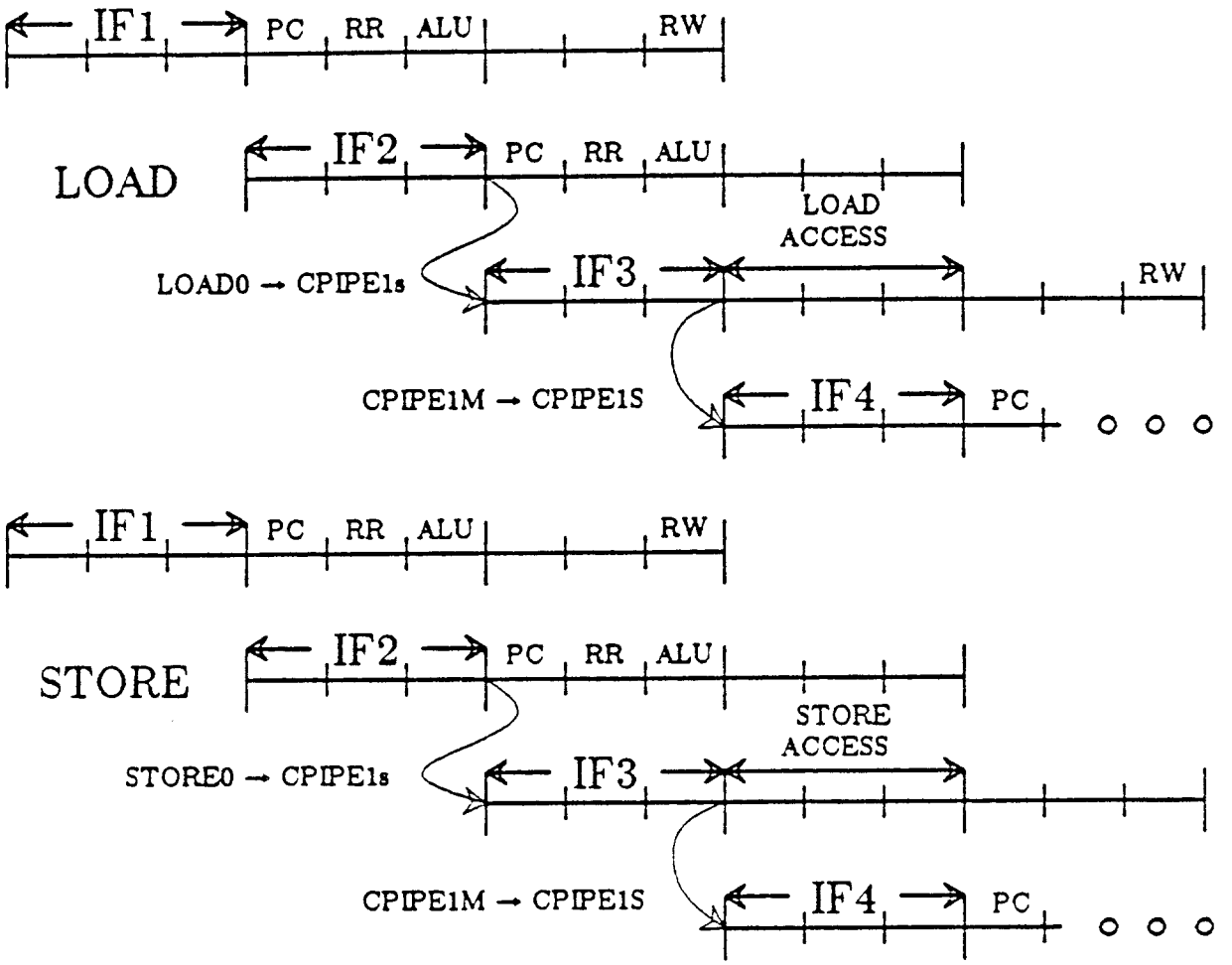


*Figure 26 Timing for Standard SOAR Load and Store*

instruction has been loaded into CPIPE1m. However, at the beginning of the **load**'s memory access cycle (it's third cycle), I3 is *not* clocked into CPIPE1s: it is held in CPIPE1m for the entire cycle. Instead of receiving CPIPE1m, CPIPE1s receives one of the internal opcodes (LOAD0). It is this opcode that will become I3, and the old I3 waiting in CPIPE1m becomes I4. The new I3, LOAD0, is the instruction that stores the loaded data into the proper register in its third cycle. I4 (the old I3) is not interrupted, nor is its execution affected in any way: it is simply forced to wait one cycle. So instead of a new instruction being fetched during IF4, the waiting instruction is allowed to move to CPIPE1s.

A **store** works analogously. Instead of forcing LOAD0 into CPIPE1s, another internal instruction STORE0 is used. And since there is no register write, there is nothing for the third cycle of the STORE0 to do. So, even though the **store** actually executes in three cycles, we charge the 'extra' cycle of the following instruction to the **store**.

### 7.3.5. Loadm and Storem Instructions

Shown in Figure 27 is the result of executing 'loadm (rx)1,r1' and 'storem r1,(rx)1' (i.e. only two registers are loaded/stored). These instructions work very much as described for the

normal **load** and **store**, and make use of other internal instructions, LOAD7 through LOAD0 and STORE7 through STORE0.

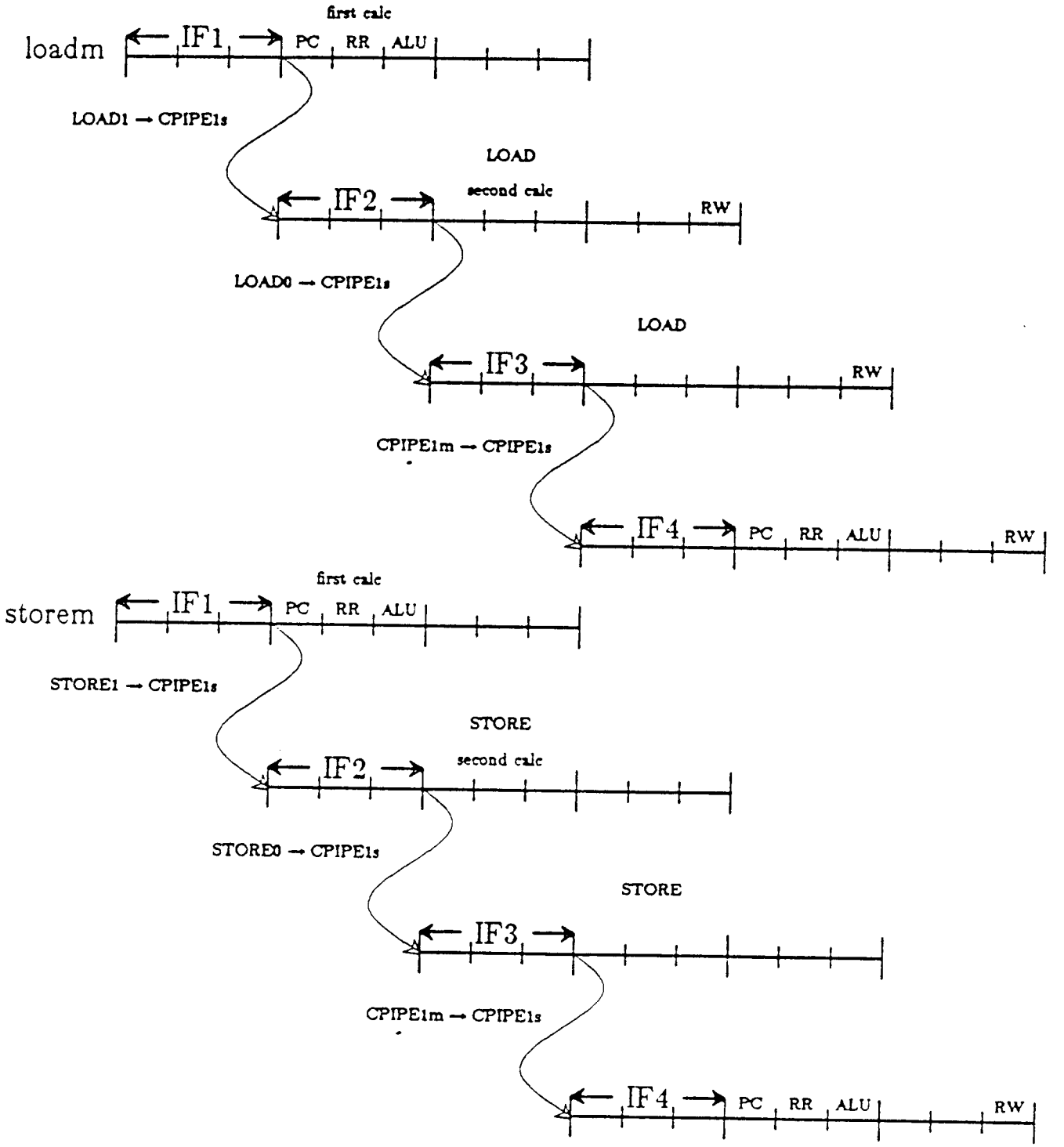


*Figure 27 Timing for Loadm and Storem*

### 7.3.6. Traps

Traps introduce new complications, but there are some general 'rules' that can help explain their impact on the pipeline. (1) If an instruction produces a trap (e.g. a tag trap), it will be manifested during that instruction's second cycle (the execution cycle when the opcode is in CPIPE1s). This is true even for instructions with illegal opcodes: their illegality is detectable during their fetch cycle, and, therefore, they have nothing to execute during the second cycle, but the trap is delayed. This principle can be considered a corollary of the more general: (2) When a trap is taken (the flow of execution is interrupted by a transfer to the trap vector), the instruction in its execution cycle (whose opcode is in CPIPE1s) is the instruction 'charged': i.e. whose opcode and operands are captured in the shadow registers.

## 7.4. Pointer-to-Register

SOAR loads and stores are conceptually different from the loads and stores of RISC I and RISC II. The 28-bit address that is the destination of a store or the origin of a load can refer to either an on-chip register or to a location in main memory. Figure 26 shows the usual timing for SOAR loads and stores, where the effective address does not refer to a register on-chip. The sequence of events is straightforward: Instruction Fetch takes all of the first cycle; the effective address computation, consisting of reading the register holding the index

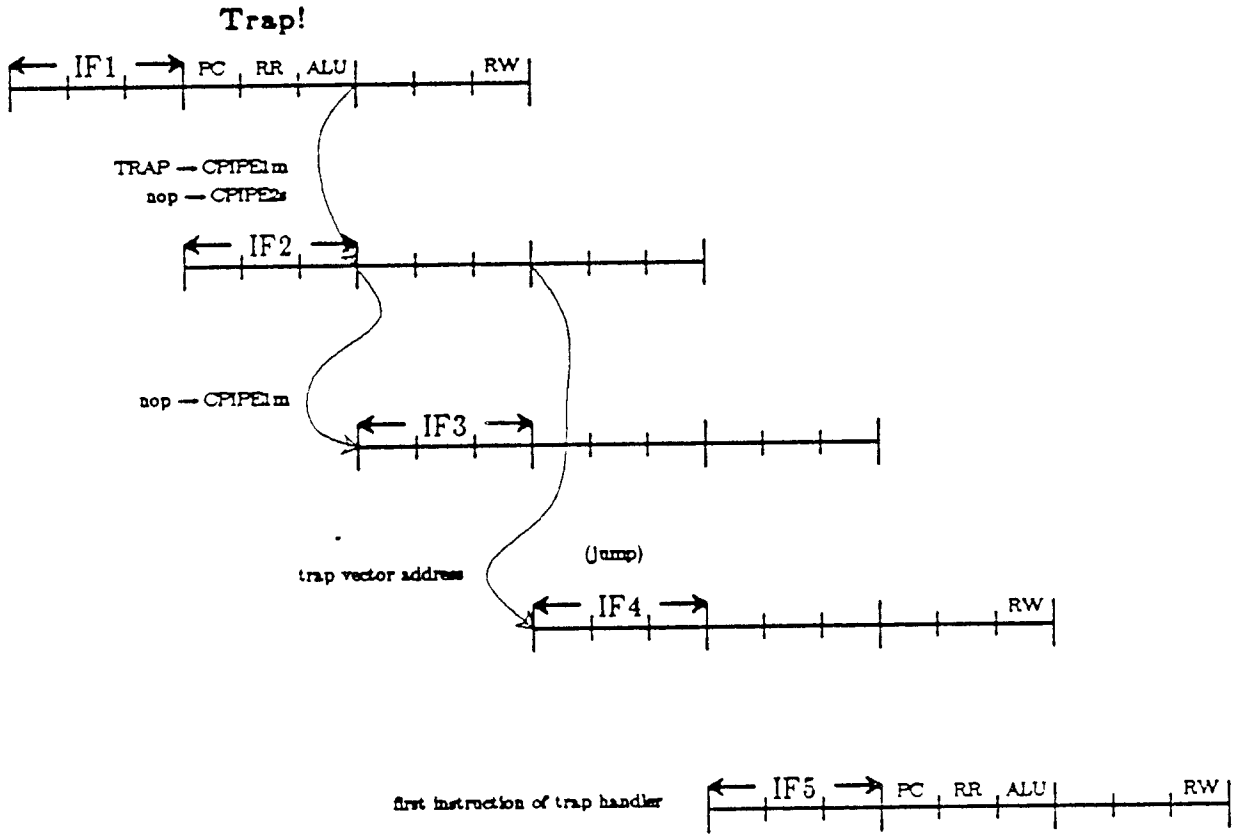


*Figure 28 Timing for Traps*

and adding the offset, occupies the second cycle; broadcasting the effective address and latching the incoming data occurs in the third cycle; and, for **load**, the final write into the on-chip register occurs in the last phase of the fourth cycle.

This scheme is complicated by the fact that the effective address may be an access to an on-chip register. The extra operations needed to accommodate this instance must occur on every **load** and every **store**, but must not slow the operation down. It is not known whether the accessed location is on-chip or not until the end of the **load/store**'s second cycle, so the check for pointer-to-register is made in PHI1 of the memory access cycle. If the **load**'s effective address is referring to a register it must be read during PHI2 of that cycle, and passed through the ALU to the destination register during PHI3. The LOAD0 internal instruction then writes the result in the appropriate register as usual without having to know where the data came from. (The LOAD0's third cycle is pictured in Figure 29 as an extension to the **load** instruction).

For pointer-to-register **stores**, much of the above is applicable. The test for pointer-to-register is done in PHI1 of the **store**'s memory access cycle. A store into a register is effected by reading the appropriate register during PHI2, and writing it in PHI3.

Since the low order seven bits of the Effective Address generated by the ALU can specify a register to be accessed on-chip, a 6-bit bus is provided between the ALU and the Register Decode logic (Address<3> is not included in the bus). The source of this bus is the DSTs. The Register Decode logic selects among three alternatives for the Register Window Number: CWP, CWP−1 (effectively), and DST<6:4> (DST<2:0> specifies the register in the window). "Flip CWP" in Figure 29 simply refers to the decision of which alternative to use.

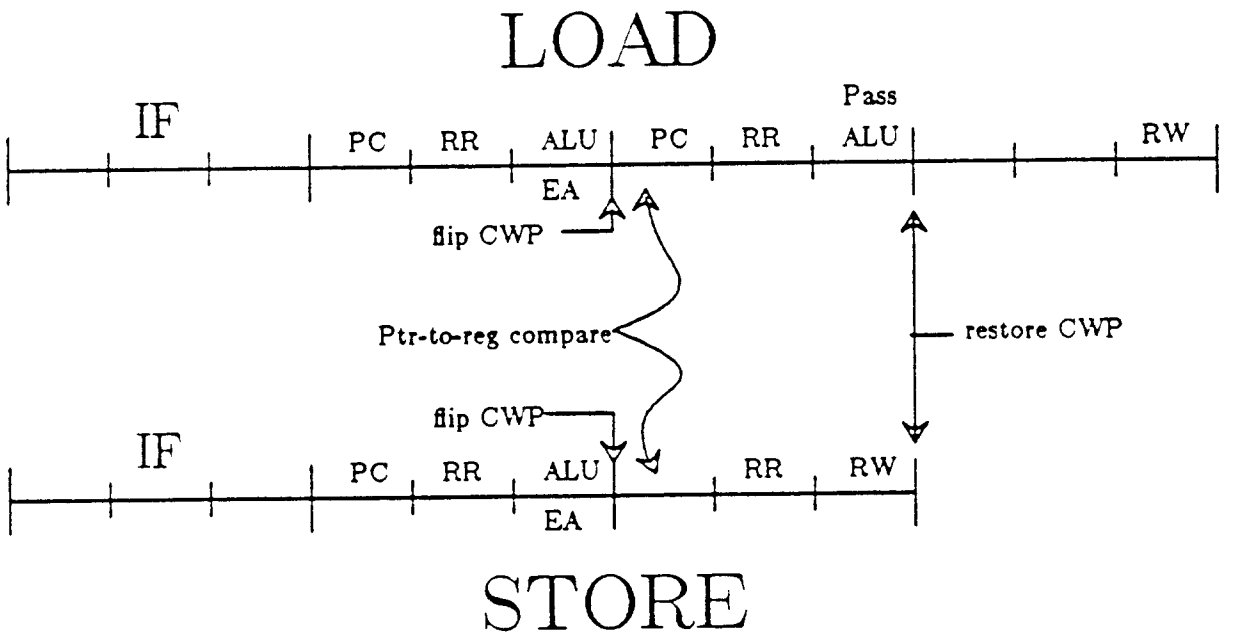


*Figure 29 Pointer-to-Register Load and Store Timing*

## 7.5. Hardware Interface

### 7.5.1. SOAR Signals

Shown in Figure 30 is a preliminary description of the signal pinout of SOAR.

| SIGNAL | PINS | TYPE | COMMENTS |
|---|---|---|---|
| D00 - D31 | 32 | Bidirectional | Data Bus |
| A00 - A31 | 32 | Output | Address Bus |
| VCC | 3 | Power | |
| GND | 3 | Ground | |
| VBB | 1 | Substrate Bias | |
| RD/WR* | 1 | Output | Read/Write Control |
| I/D* | 1 | Output | Instruction/Data fetch |
| WAIT* | 1 | Input | |
| WAITACK* | 1 | Output | |
| PHI1* | 1 | Input | Phase 1 Clock |
| PHI2* | 1 | Input | Phase 2 Clock |
| PHI3* | 1 | Input | Phase 3 Clock |
| FSHCNTL* | 1 | Output | Fast Shuffle Control |
| RESET* | 1 | Input | Reset: PSW ← 0, PC ← $0FFFFFF0_{16}$ |
| PAGE* | 1 | Input | Page Fault Interrupt |
| IO* | 1 | Input | I/O Interrupt |
| TOTAL | 82 | | |

*Figure 30 SOAR Signals*

### 7.5.2. Memory Interface

Shown in Figure 31 is the external circuitry needed to interface SOAR to memory. Some observations help to explain the reasoning behind this design.

(1) SOAR does not have a multiplexed Data/Address bus. Hence, there is no need for external address and data latches.

(2) The rate of memory accesses by the CPU is exactly once per machine cycle. Thus only a single RD/WR* control line is needed to distinguish reads from writes. pointer-to-register **loads** and **stores** are a possible exception to the 'once per machine cycle' rule, since it doesn't matter what the memory system does with the address location. Whether the real address location gets read or written is immaterial.

(3) The write strobe used by external memory should be formed from the RD/WR* line output by SOAR and the PHI3 clock signal generated by the external clock logic. It is important to use the PHI3 directly from the external clock generator, as using a write strobe generated on-chip would mean that the data and address would become invalid simultaneous with the write strobe. The illustrated scheme may run into problems if the memory has a long hold time from write invalid.

(4) External drivers must be tri-stated upon receipt of WAITACK from SOAR. The internal address bus driver will not be tri-stated during WAIT states.

### 7.5.3. Fast Shuffle™ Jumps

In the RISC design, the PC is sent out early in the cycle, and the fetched instruction is received later in that cycle. The Fast Shuffle™ mechanism examines the incoming opcode in this same cycle and, if the instruction is a **call** or **jump**, the target address is immediately loaded into the PC and used for the next instruction fetch. If a **skip** instruction is being executed while the **call** or **jump** is being fetched, the PC is not loaded with the target address if the skip condition succeeds: i.e. if the skip condition is satisfied, the **call** or **jump**

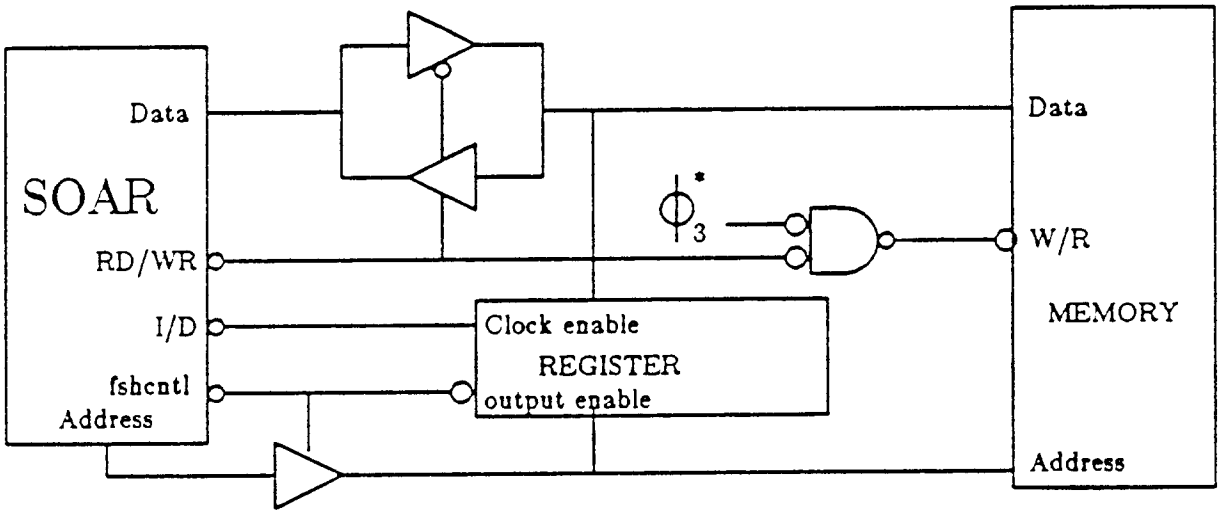


*Figure 31 External Circuitry for Memory Interface*

would not be executed.

SOAR's design allows the Fast Shuffle, but for simplicity, only provides a control signal (fshcntl) for external logic (see Figure 31). This external circuitry consists of a 28 bit register, latched to the lower 28 bits of the data bus during an instruction fetch, and a 28 bit multiplexor that drives the system address bus. SOAR's control line (FSHCNTL) is the control input to the multiplexor that selects inputs of the multiplexor. The I/D* control line is used to enable the 28 bit register. The register takes data only when I/D* is high (indicating an instruction fetch). For most cases (no Fast Shuffle), SOAR's address bus should drive the system address bus. For the Fast Shuffle case, the address comes from the external register that contains the jump address that was latched from the data bus during the previous instruction fetch.

The major problem with building all the necessary hardware on-chip is that the address coming in on the data bus during instruction fetch must in any case be able to drive the address bus pads in time for the very next cycle's instruction fetch. Chip layout considerations indicate that there is no clean way to route this data bus to address bus so that SOAR can still run at top speed. Moving the data pads very close to the address pads speeds up the Fast Shuffle bypass, but the layout for normal instruction execution is messy. Conversely, laying SOAR out like RISC I results in a large distance between the data and address pads and requires that SOAR run slower to accommodate the Fast Shuffle bypass.

### 7.6. Factors Affecting Performance of SOAR

Several factors have been identified to have significant bearings on the final level of performance of SOAR executing typical Smalltalk-80 code.

(1) The critical paths determining the cycle time of SOAR is the sum of the three phases in the second cycle and the amount of time needed to guarantee sufficient non-overlap between these cycles. Even though the typical SOAR instruction takes three cycles to execute, the second cycle is always the limiting factor since the operations performed in the first and third cycles have more time than is needed. This assumes that we have sufficiently fast external memory so that a complete memory access can be done in one SOAR cycle.

(2) The pointer-to-register scheme imposes some strict requirements on the SOAR circuitry so that cycle time is not stretched. The Effective Address and the SWP must be compared quickly, preferably by a special comparator.

(3) The critical paths of PHI1 are instruction decode and condition checking. These must both be fast. We have had to redesign our PLAs to achieve this.

(4) The critical path of PHI2 is reading the registers, driving the busses to the ALU inputs and setting up some of the ALU logic.

(5) The critical path of phase 3 is letting the carry chain settle, and then driving the Effective Address bus from the ALU into the PC and MAL.

(6) Register file decode is on the critical path. Therefore, source register decode starts in the previous PHI3 (for destination register in PHI1) to allow for maximum time for decode.

(7) For reasons of simplicity, the DST latch is of the master-slave type, which means propagation from DST latch to the Register Decode logic must be included in the register decode time (Phi1). If this propagation time proves to be significantly longer than then propagation times from the Instruction Latches to the Register Decode Logic,

then this critical path can be shortened by making the DST latch a master-only flow-through latch. (The master is used for source decoders, and the slaves are used for destination decoders. See above.)

## 8. ACKNOWLEDGEMENTS

The production of this document has depended on the talents, time, and patience of many people. Thanks to Mike Klein and Pete Foley for the original version of this document. Special thanks to Joan Pendleton for spending biillyuuns of person-moments pointing out hardware misdescriptions and correcting misconceptions about the architecture and implementation. Thanks to Dave Ungar for providing a continuing impetus to improve the document in its style and format. And also thanks (but not in the sense of "also ran") to Ricki Blau, Will Brown, Bill Bush, Paul Hilfinger, Peter Lee, and Dave Patterson for their comments and support.

## 9. REFERENCES

[Blak83]: Blakken, John, "Register Windows for SOAR," Chapter 6, *Proceedings of CS292R,* April 1983.

[Blau83]: Blau, Ricki, "Tags and Traps for the SOAR Architecture," Chapter 2, *Proceedings of CS292R,* April 1983. (see Chapter 3, "VLSI Design Considerations for SOAR").

[Blom83]: Blomseth, Richard, and Davis, Helen, "The Orion Project – A Home for SOAR," Chapter 4, *Proceedings of CS292R,* April 1983.

[Bos83]: Bose, B.K., Mattausch, Hans-Juergen, und Schallenberger, Burgahrdt, "VLSI Design Considerations for SOAR," Chapter 3, *Proceedings of CS292R,* April 1983.

[Citr83]: Citrin, Wayne, and Ponder, Carl, "Implementing a Smalltalk Compiler," Chapter 9, *Proceedings of CS292R,* April 1983.

[DAmb83]: D'Ambrosio, Bruce, "Smalltalk-80 Language Measurements - Dynamic Use of Compiled Methods," Chapter 5, *Proceedings of CS292R,* April 1983.

[Kate83]: Katevenis, M.G.H, *Reduced Instruction Set Computer Architectures for VLSI,* Ph.D. Dissertation, Computer Science (EECCS), UC Berkeley, Report No. UCB/CSD 83/141. (particularly pp.54ff)

[Laru83]: Larus, James, and Bush, William, "Classy: A Method for Efficiently Compiling Smalltalk," Chapter 10, *Proceedings of CS292R,* April 1983.

[Patt83]: Patterson, Dave, "Second SOAR," February 1983.

[Samp84]: Samples, A. Dain, "Daedalus: Software for SOARing on a Sun," (in process).

[Unga83a]: Ungar, Dave, "A Proposal for SOAR Interrupts and Traps," April 1983.

[Unga83b]: Ungar, Dave, "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm."

## 10. APPENDIX A: Memory addresses of windows

Question: what is the memory address of the context object containing a specific window?

Corollary question: how does changing the SWP and/or CWP change the address computation?

Assume that SWP is initially F00 (all numbers here are in hex, and all memory diagrams have location zero at the top). Then the (present and future) addresses of the windows are:

| | |
|---|---|
| : | : |
| E40 | W4 |
| E50 | W5 |
| E60 | W6 |
| E70 | W7 |
| E80 | W0 |
| E90 | W1 |
| EA0 | W2 |
| EB0 | W3 |
| EC0 | W4 |
| ED0 | W5 |
| EE0 | W6 |
| EF0 | W7 |
| F00 | : |

The table below shows the addresses of the windows at various values of SWP. In other words, if the SWP is now EE0, then the current address of window 2, that is, where window 2 will be stored on overflow, is EA0.

| | window | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| SWP= | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| F00 | XXX | E90 | EA0 | EB0 | EC0 | ED0 | EE0 | EF0 |
| EF0 | E80 | E90 | EA0 | EB0 | EC0 | ED0 | EE0 | XXX |
| EE0 | E80 | E90 | EA0 | EB0 | EC0 | ED0 | XXX | E70 |
| ED0 | E80 | E90 | EA0 | EB0 | EC0 | XXX | E60 | E70 |
| EC0 | E80 | E90 | EA0 | EB0 | XXX | E50 | E60 | E70 |
| EB0 | E80 | E90 | EA0 | XXX | E40 | E50 | E60 | E70 |
| EA0 | E80 | E90 | XXX | E30 | E40 | E50 | E60 | E70 |
| E90 | E80 | XXX | E20 | E30 | E40 | E50 | E60 | E70 |
| E80 | XXX | E10 | E20 | E30 | E40 | E50 | E60 | E70 |
| E70 | E00 | E10 | E20 | E30 | E40 | E50 | E60 | XXX |
| : | : | : | : | : | : | : | : | : |

Note that the XXX entries cannot happen and do not concern us. That is, SWP<6:4> can never equal CWP<6:4> unless the two registers were initialized incorrectly. Consider: that would mean that the last window saved, window SWP<6:4>, occupied the window that is currently the HIGH registers of the executing procedure. Then either the window was just saved by the window overflow handler, in which case the SWP would have been decremented by the call, or it was just restored/reloaded by the window underflow handler, in which case

the SWP would have been incremented by the ret. In neither case will SWP<6:4>=CWP<6:4>. Of course, this will happen briefly during the execution of the window under-/overflow trap handlers. But after all, they are EXCEPTION handlers. No user code should ever be in this state.

Another way of looking at it is that call instructions 'push down' the value of SWP<6:4> when necessary so that it is always at least one less than CWP<6:4>. Return instructions always 'push up' the value of SWP<6:4> when necessary so that it is always at most one more than CWP<6:4>. When the SWP must be 'pushed', a window over- or underflow trap occurs. At no time do the return or call instructions allow SWP<6:4>=CWP<6:4>.

From the discussion above, it is easy to confirm that the address A(i) of window i is:

$$A(i) = SWP - ( ( SWP - (i * 10) ) \& 70 ) \qquad 0 <= i <= 7$$

(remember that all numbers are in hex). If one is interested in the address of the current window, use CWP in place of (i * 10) in the expression. The SOAR code for this computation would look like that given in section §5.

## 11. APPENDIX B: Interaction of Windows, SWP, and CWP

The interaction of the register file, CWP, and SWP is not difficult to grasp intuitively. Getting all of the numbers to work out in detail, however, is a wheel of a different magnitude. This appendix will hopefully prevent others from having to reinvent this wheel. (Also, see [Kate83] for an alternative explanation in the context of RISC II.)

Assume the CWP is initialized to 7, and the SWP is initialized to some value, the last seven bits of which are zero (note below that only the last eight bits are displayed: x is a binary digit and w is the other binary digit). Assume that we have a root program P1 with procedures P2, P3, ...

| ACTION | CWP (after action) | SWP (after action) | on-chip windows w7 | w6 | w5 | w4 | w3 | w2 | w1 | w0 |
|---|---|---|---|---|---|---|---|---|---|---|
| initialize | 7 | x000 0000 | 1H | 1L | – | – | – | – | – | – |
| P1 calls P2 | 6 | x000 0000 | 1H | 2H | 2L | – | – | – | – | – |
| P2 calls P3 | 5 | x000 0000 | 1H | 2H | 3H | 3L | – | – | – | – |
| P3 calls P4 | 4 | x000 0000 | 1H | 2H | 3H | 4H | 4L | – | – | – |
| P4 calls P5 | 3 | x000 0000 | 1H | 2H | 3H | 4H | 5H | 5L | – | – |
| P5 calls P6 | 2 | x000 0000 | 1H | 2H | 3H | 4H | 5H | 6H | 6L | – |
| P6 calls P7 | 1 | x000 0000 | 1H | 2H | 3H | 4H | 5H | 6H | 7H | 7L |
| P7 calls P8, trap | 1 | x000 0000 | 1H | 2H | 3H | 4H | 5H | 6H | TH | TL |
| decr SWP and CWP, store w7 (1H) into M[w111 1000] thru M[w111 1111] | | | | | | | | | | |
| | 0 | w111 0000 | 8L | 2H | 3H | 4H | 5H | 6H | 7H | 8H |
| P8 calls P9, trap | 7 | w111 0000 | TL | 2H | 3H | 4H | 5H | 6H | 7H | TH |
| decr SWP and CWP, store w6 (2H) into M[w110 1000] thru M[w110 1111] | | | | | | | | | | |
| | 7 | w110 0000 | 9H | 9L | 3H | 4H | 5H | 6H | 7H | 8H |
| P9 calls Pa, trap | 6 | w110 0000 | TH | TL | 3H | 4H | 5H | 6H | 7H | 8H |
| decr SWP and CWP, store w5 (3H) into M[w101 1000] thru M[w101 1111] | | | | | | | | | | |
| | 6 | w101 0000 | 9H | aH | aL | 4H | 5H | 6H | 7H | 8H |
| Pa returns to P9 | 7 | w101 0000 | 9H | 9L | – | 4H | 5H | 6H | 7H | 8H |
| P9 returns to P8 | 0 | w101 0000 | 8L | – | – | 4H | 5H | 6H | 7H | 8H |
| P8 returns to P7 | 1 | w101 0000 | – | – | – | 4H | 5H | 6H | 7H | 7L |
| P7 returns to P6 | 2 | w101 0000 | – | – | – | 4H | 5H | 6H | 6L | – |
| P6 returns to P5 | 3 | w101 0000 | – | – | – | 4H | 5H | 5L | – | – |
| P5 returns to P4 | 4 | w101 0000 | – | – | – | 4H | 4L | – | – | – |
| P4 returns to P3, trap | 4 | w101 0000 | – | – | – | TH | TL | – | – | – |
| read M[w101 1000] thru M[w101 1111] (3H) into w5, incr SWP and CWP | | | | | | | | | | |
| | 5 | w110 0000 | – | – | 3H | 3L | – | – | – | – |
| P3 returns to P2, trap | 5 | w110 0000 | – | – | TH | TL | – | – | – | – |
| read M[w110 1000] thru M[w110 1111] (2H) into w6, incr SWP and CWP | | | | | | | | | | |
| | 6 | w111 0000 | – | 2H | 2L | – | – | – | – | – |
| P2 returns to P1, trap | 6 | w111 0000 | – | TH | TL | – | – | – | – | – |
| read M[w111 1000] thru M[w111 1111] (1H) into w7, incr SWP and CWP | | | | | | | | | | |
| | 7 | x000 0000 | 1H | 1L | – | – | – | – | – | – |
| And we are back to the original state. Of course, any more returns after this point would result in a major system error. | | | | | | | | | | |

## 12. APPENDIX C: Available Integer Constants

At first glance it may appear that only integers in the range -128 to +127 can be represented in immediate fields: not so. Remember that the upper four bits are not discarded, simply shifted up to the high end. Here is an exhaustive list of the ranges of 32-bit untagged integers representable in the 12 bits of the immediate fields.

| | | | | |
|---|---|---|---|---|
| -2147483648 | (0x8000000) | ... | -2147483521 | (0x800007f) |
| -1879048320 | (0x8fff80) | ... | -1879048065 | (0x900007f) |
| -1610612864 | (0x9fff80) | ... | -1610612609 | (0xa00007f) |
| -1342177408 | (0xafff80) | ... | -1342177153 | (0xb00007f) |
| -1073741952 | (0xbfff80) | ... | -1073741697 | (0xc00007f) |
| -805306496 | (0xcfff80) | ... | -805306241 | (0xd00007f) |
| -536871040 | (0xdfff80) | ... | -536870785 | (0xe00007f) |
| -268435584 | (0xefff80) | ... | -268435329 | (0xf00007f) |
| -128 | (0xffff80) | ... | 127 | (0x000007f) |
| 268435328 | (0x0fff80) | ... | 268435583 | (0x100007f) |
| 536870784 | (0x1fff80) | ... | 536871039 | (0x200007f) |
| 805306240 | (0x2fff80) | ... | 805306495 | (0x300007f) |
| 1073741696 | (0x3fff80) | ... | 1073741951 | (0x400007f) |
| 1342177152 | (0x4fff80) | ... | 1342177407 | (0x500007f) |
| 1610612608 | (0x5fff80) | ... | 1610612863 | (0x600007f) |
| 1879048064 | (0x6fff80) | ... | 1879048319 | (0x700007f) |
| 2147483520 | (0x7fff80) | ... | 2147483647 | (0x7fffff) |

Equivalent values when the immediates are 31-bit tagged integers:

| | | | | |
|---|---|---|---|---|
| -1073741824 | (0x4000000) | ... | -1073741697 | (0x400007f) |
| -805306496 | (0x4fff80) | ... | -805306241 | (0x500007f) |
| -536871040 | (0x5fff80) | ... | -536870785 | (0x600007f) |
| -268435584 | (0x6fff80) | ... | -268435329 | (0x700007f) |
| -128 | (0x7fff80) | ... | 127 | (0x000007f) |
| 268435328 | (0x0fff80) | ... | 268435583 | (0x100007f) |
| 536870784 | (0x1fff80) | ... | 536871039 | (0x200007f) |
| 805306240 | (0x2fff80) | ... | 805306495 | (0x300007f) |
| 1073741696 | (0x3fff80) | ... | 1073741823 | (0x3fffff) |

## 13. APPENDIX D: Trap Vector Assignments

These tables show which traps can occur on which instructions by listing them as they appear in the trap vector. The first column of each table is the number of the trap appended to the opcode of the interrupted instruction.

| trap: ILL | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 0000-0003 | * | illegalOpcode | |
| 0004 | nop | illegalOpcode31 | *I<31>=1* |
| 0005 | (TRAP) | illegalOpcode31 | *I<31>=1* |
| 0006 | (SKIP) | illegalOpcode31 | *I<31>=1* |
| 0007 | | illegalOpcode | |
| 0010 | ret | illegalOpcode31 | *I<31>=1* |
| 0011 | retw | illegalOpcode31 | *I<31>=1* |
| 0012 | retn | illegalOpcode31 | *I<31>=1* |
| 0013 | retnw | illegalOpcode31 | *I<31>=1* |
| 0014 | reti | illegalOpcode31 | *I<31>=1* |
| 0015 | retiw | illegalOpcode31 | *I<31>=1* |
| 0016 | retin | illegalOpcode31 | *I<31>=1* |
| 0017 | retinw | illegalOpcode31 | *I<31>=1* |
| 0020 | skip | illegalOpcode31 | *I<31>=1* |
| 0021-0027 | trapi | illegalOpcode31 | *I<31>=1* |
| 0030 | store | illegalOpcode31 | *I<31>=1* |
| 0031 | | illegalOpcode | |
| 0032 | storem | illegalOpcode31 | *I<31>=1* |
| 0033 | | illegalOpcode | |
| 0034 | load | illegalOpcode31 | *I<31>=1* |
| 0035 | loadc | illegalOpcode31 | *I<31>=1* |
| 0036 | loadm | illegalOpcode31 | *I<31>=1* |
| 0037 | | illegalOpcode | |
| 0040 | srl | illegalOpcode31 | *I<31>=1* |
| 0041 | | illegalOpcode | |
| 0042 | sra | illegalOpcode31 | *I<31>=1* |
| 0043 | | illegalOpcode | |
| 0044 | xor | illegalOpcode31 | *I<31>=1* |
| 0045 | | illegalOpcode | |
| 0046 | and | illegalOpcode31 | *I<31>=1* |
| 0047 | or | illegalOpcode31 | *I<31>=1* |
| 0050 | add | illegalOpcode31 | *I<31>=1* |
| 0051 | sla | illegalOpcode31 | *I<31>=1* |
| 0052 | sub | illegalOpcode31 | *I<31>=1* |
| 0053 | | illegalOpcode | |
| 0054 | insert | illegalOpcode31 | *I<31>=1* |
| 0055 | | illegalOpcode | |
| 0056 | extract | illegalOpcode31 | *I<31>=1* |
| 0057 | | illegalOpcode | |
| 0060-0067 | (LOADi) | illegalOpcode31 | *I<31>=1* |
| 0070-0077 | (STOREi) | illegalOpcode31 | *I<31>=1* |

| trap: TT | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 0100-0117 | | * | *can't happen* |
| 0120 | skip | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0121 | trap1 | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0122 | trap2 | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0123 | trap3 | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0124 | trap4 | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0125 | trap5 | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0126 | trap6 | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0127 | trap7 | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0130 | store | storeTagTrap | *see Table 14* |
| 0131-0133 | | * | *can't happen* |
| 0134 | load | loadTagTrap1 | *see Table 13* |
| 0135 | loadc | SITagTrap | *see Table 13* |
| 0136-0137 | | * | *can't happen* |
| 0140 | srl | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0141 | | * | *can't happen* |
| 0142 | sra | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0143 | | * | *can't happen* |
| 0144 | xor | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0145 | | * | *can't happen* |
| 0146 | and | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0147 | or | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0150 | add | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0151 | sia | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0152 | sub | aluTagTrap | *(S1=oop \|\| S2=oop)* |
| 0153-0177 | | * | *can't happen* |

| trap: SWI | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 0200-0237 | call | SWITrap | *tagged call and PSW<5>=1* |
| 0240-0277 | jump | SWITrap | *tagged jump and PSW<5>=1* |

| trap: WO | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 0300-0337 | call | WOFTrap | *(cf. §4.2.6)* |
| 0340-0377 | | * | *can't happen* |

| trap: WU | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 0400-0407 | | * | *can't happen* |
| 0410 | ret | * | *can't happen* |
| 0411 | retw | WUFTrap | *(cf. §4.2.7)* |
| 0412 | retn | * | *can't happen* |
| 0413 | retnw | WUFTrap | *(cf. §4.2.7)* |
| 0414 | reti | * | *can't happen* |
| 0415 | retiw | WUFTrap | *(cf. §4.2.7)* |
| 0416 | retin | * | *can't happen* |
| 0417 | retinw | WUFTrap | *(cf. §4.2.7)* |
| 0420-0477 | | * | *can't happen* |

| trap: DPF | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 0500-0527 | | * | *can't happen* |
| 0530 | store | dataPageFlt | *page fault* |
| 0531 | | * | *can't happen* |
| 0532 | storem | dataPageFlt | *page fault* |
| 0533 | | * | *can't happen* |
| 0534 | load | dataPageFlt | *page fault* |
| 0535 | loadc | dataPageFlt | *page fault* |
| 0536 | loadm | dataPageFlt | *page fault* |
| 0537-0577 | | * | *can't happen* |

| trap: TI | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 0600-0620 | | * | *can't happen* |
| 0621 | trap1 | trap1Instruction | *operands satisfy condition* |
| 0622 | trap2 | trap2Instruction | *operands satisfy condition* |
| 0623 | trap3 | trap3Instruction | *operands satisfy condition* |
| 0624 | trap4 | trap4Instruction | *operands satisfy condition* |
| 0625 | trap5 | trap5Instruction | *operands satisfy condition* |
| 0626 | trap6 | trap6Instruction | *operands satisfy condition* |
| 0627 | trap7 | trap7Instruction | *operands satisfy condition* |
| 0630-0677 | | * | *can't happen* |

| trap: GS | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 0700-0707 | | * | *can't happen* |
| 0710 | ret | gcTrap | *S1=oop* |
| 0711 | retw | gcTrap | *S1=oop* |
| 0712 | retn | gcTrap | *S1=oop* |
| 0713 | retnw | gcTrap | *S1=oop* |
| 0714 | reti | gcTrap | *S1=oop* |
| 0715 | retiw | gcTrap | *S1=oop* |
| 0716 | retin | gcTrap | *S1=oop* |
| 0717 | retinw | gcTrap | *S1=oop* |
| 0720-0727 | | * | *can't happen* |
| 0730 | store | gcTrap | *S2 older than S1, context* |
| 0731-0777 | | * | *can't happen* |

| trap: IPF | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 1000-1077 | (any) | instrPageFlt | *page fault* |

| trap: IO | | | |
|---|---|---|---|
| tb+(octal) | opcode | handler | happens when ... |
| 1100-1177 | (any) | IOInterrupt | *I/O interrupt* |

## 14. APPENDIX E: Caveats

This section summarizes the "caveats" and some of the surprising features of the architecture. We will also try to give some rationale for why things are the way they are.

**Do not use the SPECIAL registers in field S2; do not use a SPECIAL register in the S1 field of an instruction when the S2 field requires register forwarding.** In general, the register designated by the S1 field of an instruction is put on the A-bus (refer to Figure 22), and the register designated in the S2 field is put on the B-bus. The S2 field may not designate a special register because there is no way to get some of the special registers (particularly CWP and PSW) onto the B-bus. The reason the other special registers (SWP, TB, PC, SHA, and SHB) cannot be put on the B-bus via the D-bus is that the result of the previous instruction is placed on the D-bus to find its way to the A-bus or B-bus, as appropriate, when register forwarding occurs. Hence, none of the special registers may be put in the S2 field of an instruction. Moreover, putting a special register in the S1 field when the S2 field requires register forwarding causes the forwarded value from DSTs and the value of the PC, SWP, or TB registers to compete for the D-bus. (Actually, only PC, SWP and TB cannot be used in S1 under these circumstances, since SHA, SHB, CWP, and PSW can safely go to the A-bus.) The following is an example of what is prohibited.

```
add     r10,r11,r6      ;tagged add
add     SWP,r6,r5       ;r6 requires forwarding,
                        ; conflicts with SWP
```

**Do not use the SPECIAL registers in the destination field of load instructions.**

**Do not write to the PC register.**

**Do not assume the memory system ignores bits 28 through 31 in the Trap Base register.**

**Bit 12, the immediate bit, of store instructions must be = 1.**

**The destination field D of return instructions must be zero.**

**Do not read PSW<15:8>, SHA, and SHB with interrupts enabled.** PSW<15:8>, registers SHA, and SHB contain valid values only if interrupts are disabled. Their contents are undefined if read while interrupts are enabled. This is because they are implemented as flow-through latches, not as registers. Therefore, if one tries to read them when interrupts are enabled, they would be shadowing at the same time. The remainder of the PSW register behaves as defined in §2.2.

**Do not attempt to read r0 after a loadm: wait one cycle.**

**Do not attempt to use the internal opcodes in the instruction stream.**