1. 386 LOADALL Instruction

The 80386 implements a LOADALL instruction capable of loading all visible machine state from an area in memory. The function of this instruction is the same as the 80286 LOADALL instruction [RASH]. Due to the architecture, and implementation of the 386, the format of the memory area used by the two processors is different. Conversion between these formats is possible, requiring an algorithm to 'map' between the two formats, and compensate for the architectural differences.

The attached tables illustrate the memory area used by LOADALL. The location of this area in memory is dynamic, not fixed at 800H as in the 80286. The LOADALL instruction uses ES:EDI for the base of the LOADALL memory image. Note that the normal address computation scheme applies to LOADALL; if protection is enabled (CR0.PE), and possibly paging is enabled (CR0.PG), the physical address of the memory area will be computed by both segment, and page relocation. A typical sequence to perform the LOADALL would be:

```
MOV    AX, Dump_Area_Selector
MOV    ES, AX
MOV    EDI, offset Dump_Area
LOADALL
```

1.1. Background

As part of marketing the 286, most of the details of one of the 286 test instructions were published to selected customers and ISVs in a note entitled "iAPX 286 LOADALL Instruction", by Bill Rash. The test instruction was named LOADALL for public consumption. The instruction was published to provide two functions: to allow REAL mode code to access memory above 1 Meg, and to allow a protected mode OS to simulate 8086 semantics for segment register loads as a key part of implementing a "virtual 8086 mode" on the 286. When the details of the "LOADALL" instruction were released, we made it quite clear that the instruction would not be supported on the 386. However, regardless of our warnings, by publishing the details of the 286 LOADALL we effectively made it a part of the 286 architecture. People will use the instruction anyway, and we must support the 286 LOADALL instruction on the 386.

The 386 has similar test instructions, which are "culturally compatible" with the 286 test instructions. The instructions on both machines reload internal machine state from a block of memory. However, the format of the memory area is quite different, and the 386 LOADALL loads more state. Things that were 16-bits on the 286 were stretched to 32 bits on the 386.

1.2. Emulating 286 LOADALL with 386 LOADALL

As noted above, the 386 LOADALL function is pretty close the the 286 LOADALL instruction, and it is possible to reformat the 286 image to a 386 image and use the 386 instruction to provide emulation capability. This requires the following:

o The 386 must trap the 286 LOADALL opcode.

o Need code to translate 286 LOADALL format to 386 format.

The 386 traps the 286 LOADALL opcode, since the 286 LOADALL opcode (0F05) an illegal opcode on the 386. The 386 LOADALL has a different opcode (0F07). A 386 OS that wishes to emulate the 286 LOADALL can include an interrupt 6 (invalid opcode fault) handler to emulate the 286 LOADALL. The invalid opcode fault handler will be invoked whenever the 286 LOADALL instruction occurs in the instruction stream. The fault handler can decode the invalid instruction, and if it is the 286 LOADALL, can reformat the memory image in a new area, and execute a 386 LOADALL. Because the 386 LOADALL can load from an arbitrary address, the block at 800H can be left undisturbed, with the reformatting done in a part of the address space not accessible on the 286 (above 16 Meg, for example).

The information loaded by the 286 LOADALL falls into 3 categories: programmer visible registers, 286 specific temp registers, and "invisible" descriptor cache registers. There is no problem with emulating the loading of the programmer visible registers, as these are compatibly implemented on the 386 (and the 486, 586, ...). The 286 also loads a number of temp registers, but the values in these registers are "dead" when the next instruction (other than STOREALL) begins execution. Consequently, the values loaded into the temp registers can have no effect on 286 program execution, so these can be ignored. As long as the 386 temps are also "dead" when the next instruction begins (except for STOREALL), there will be no problems with 386-specific temp registers.

In order to ease the burden of supporting LOADALL on future processors, the temp register dump/load area is at the top of the LOADALL block, so it can grow and shrink as required for future implementations. Several difficulties occur when attempting to emulate the loading of the "invisible" descriptor cache registers. Unfortunately, this is the main reason why the 286 loadall is an interesting instruction...

To verify that the 386 LOADALL can emulate the 286 LOADALL, we need to verify that all of the "invisible" 286 descriptor cache entries modified by the 286 LOADALL are also modified by the 386 LOADALL. Any extra state can just be loaded with the 286 compatible values. Some "don't care" values on the 286 are now recognized by the 386, and so these may need to be parsed and reformatted.

1.2.1. Translating 286 LOADALL format to 386 LOADALL

This section sketches the algorithm for translating the 286 LOADALL format to the format required for 386 LOADALL. Special considerations for each field in the 286 descriptor cache are given below:

MSW

Only the lower 4 bits are defined on the 286. Bit 0 is sticky on the 286 LOADALL, so software must OR in the current setting of bit 0 of CR0 with the 286 LOADALL image to get the CR0 image for the 386 LOADALL. The ET bit is new on the 386, as is the PG bit. Both of these bits should remain unchanged, and so should be copied from the current CR0 value. This can be done with the following code sequence:

```
MOV  EAX, CR0
AND  EAX, 80000011H
MOV  CX, LD286.MSW  ; Load 286 MSW image
AND  ECX, 0Fh       ; mask low bits
OR   EAX, ECX
MOV  LD386.CR0      ; value for 386 LOADALL
```

TR, LDT, DS, SS, CS, ES Selectors

These can be copied directly.

FLAGS

The 386 defines two new flags. The 286 flags can be copied directly to the low order 16 bits of the 386 flags. The RF bit can be set to 0, and the VM bit set if the emulated program is in Virtual 8086 mode (if the LOADALL was trapped in a Virtual 8086 program), and cleared otherwise. The VM bit can be copied from the EFLAGS image pushed when the invalid opcode trap is taken.

IP

The 286 IP register is copied to the low order 16 bits of the 386 EIP image, and the upper 16 bits of the EIP image cleared.

AX..SI

The 286 register images are copied to the lower 16 bits of the 386 32-bit registers. The upper 16 bits can be random trash, or could be set to 0 for tidyness. Only EIP needs to have its upper bits cleared.

ES..DS Descriptor Cache Entries

These can be reformatted to 386 descriptor cache entries. The base and limit values translate directly to the 386, with extra high-order zeros. However, the AR values may not translate.

The 286 uses only bit 1, bit 2, bit 3, and bit 7 of the AR byte to do protection checks. The 386 uses these bits, plus the "G" and "B" bits to perform checks. If the "G" and "B" bits are 0, the checks are 100% compatible. The 286 and 386 may behave differently if the DPL field of the SS and CS descriptor entries are not equal, or if the RPL fields of the SS and CS selectors do not match the DPL field of the SS and CS descriptors. The 286 action in this case is described as "undefined" in the LOADALL description. CPL is loaded from the AR byte for the SS register (SS, not CS). If LOADALL is executed in protected mode, errors may occur in subsequent instructions if the RPL field of SS or CS selectors, or the DPL of the CS descriptor, do not match the DPL field of the SS descriptor.

GDTR, IDTR Registers

These translate directly. The low 5 bytes of the 286 LOADALL image are moved to the 386 image, and the upper byte is set to 0.

LDT Descriptor Cache

Similar to other descriptor cache entries, except that only a subset of the AR bits are on the 286 and 386. Only the P bit is supported on both machines, so this descriptor entry should translate directly.

TSS Descriptor Cache

Similar to other descriptor cache entries, except for the treatment of AR bits again. The 286 recognizes none of the AR bits. The 386 recognizes bit 3 to distinguish 286 TSS types from 386 TSS types. The 286 descriptor can be copied to the 386 LOADALL image directly, with bit 3 of the AR field set to 0.

To summarize, the 386 LOADALL instruction can be used to emulate the 286 LOADALL, except for some questionable areas involving the Access Rights bytes in the "hidden" descriptor entries. The 286 LOADALL can be used to load these AR fields with inconsistant values, in which case we don't know what the 286 will do, let alone whether the 386 matches the 286 semantics. The 286 LOADALL can be emulated by the 386 LOADALL provided that the

DPL and RPL values of CS and SS are all equal. Other cases are undefined by the 286 (and cannot be induced by executing normal instructions).

1.3. LOADALL used to switch Modes

Any of the 80386 operating modes may be selected with LOADALL. The PE (Protection Enabled) bit is not 'Sticky' as in the 286. By setting the appropriate bits in the memory area, the processor will resume execution in the selected mode after LOADALL. For example, a LOADALL could be performed to a page protected VM86 task by setting the following conditions in the memory image:

PE and PG bits set to 1
VM bit set in the extended flags register
8086 style segment register values into the memory image
Descriptor bases to (Segment Reg SHL 4)
Access rights= (Present,ByteGranular etc.)
LIMIT set to 0000FFFFH.

As another example, RESET can be emulated with LOADALL. By loading The processors initial values into the memory image as follows:

All registers=0
GS,FS,DS,ES,SS=0
CS=F000, EIP=FFF0, CS-Base=FFFF0000
GS,FS,DS,ES,SS-Base=0
GS,FS,DS,ES,SS-AR= Present
GS,FS,DS,ES,SS-Limit = FFFF
CR0=0
All other control registers=0
Debug Registers=0
EFLAGS=0

After the LOADALL instruction, the processor state will be identical to reset. could be set to run at a CPL <> 0 while in real mode. Combinations such as these are possible, but may have unexpected results. Placing the processor into other than a 'Natural' state should be avoided.

1.4. Caveats

Note that LOADALL provides no error checking. It is important that the descriptor entries match the selector entries (Unless the intent is that they do not, for example to give a REAL mode program access to the extended address space). After LOADALL, the CPL of the processor will be set to the DPL of the SS Descriptor entry. This is necessary to accomodate conforming segments. The IOPL will be set from the IOPL value in the EFLAGS image.

LOADALL is not restartable. If a page or segment fault occurs during execution of LOADALL, the processor will be left in an undefined state. LOADALL is a privileged instruction, so it can be executed only at level 0. It is assumed that the OS kernal will restrict use of this instruction, and will check for addressability and for consistent semantics before executing the LOADALL.

1.5. 386 LOADALL Memory Format

The following tables define the LOADALL memory format. The LOADALL instruction uses a 512-byte block of memory, where the lowest addressed byte is given in ES:[(E)DI]. The area above offset CC hex is used for processor dependent registers (temporaries, invisible registers). These are loaded into the processor, but will not affect normal program execution. All values in the memory area are read from a four byte field, to keep the memory format DWORD Aligned, but it is possible to locate memory area at a non-aligned address. In this case, the execution time of LOADALL will DOUBLE! For this reason, the memory dump area should always be D-Word aligned.

Each descriptor entry consists of 3 pieces:

AR
Base
Limit

The AR part has the same format as the second dword of a segment descriptor except that only the AR byte (bits 8-15) and the G and B/D bits (bits 23 and 22) are used. All other bits in the AR field are ignored. The Base and Limit parts contain full 32-bit values, fully expanded and unscrambled from the 386 descriptor. In particular, the Limit field loaded for a page granular segment gives a byte granular limit, so should contain the page limit*4096 plus 4095.

80386 LOADALL memory format	
User Register Area	
Offset	Register
000	CR0
004	EFLAGS
008	EIP
00C	EDI
010	ESI
014	EBP
018	ESP
01C	EBX
020	EDX
024	ECX
028	EAX
02C	DR6
030	DR7
034	TSSR(TSSSelector-Word)
038	LDTR(LDTSelector-Word)
03C	GS
040	FS
044	DS
048	SS
04C	CS
050	ES

80386 LOADALL memory format	
Segment Descriptor Area	
Offset	Register
054	TSS(AR)
058	TSS(BASE)
05C	TSS(LIMIT)
060	IDT(AR)
064	IDT(BASE)
068	IDT(LIMIT)
06C	GDT(AR)
070	GDT(BASE)
074	GDT(LIMIT)
078	LDT(AR)
07C	LDT(BASE)
080	LDT(LIMIT)
084	GS(AR)
088	GS(BASE)
08C	GS(LIMIT)
090	FS(AR)
094	FS(BASE)
098	FS(LIMIT)
09C	DS(AR)
0A0	DS(BASE)
0A4	DS(LIMIT)
0A8	SS(AR)
0AC	SS(BASE)
0B0	SS(LIMIT)
0B4	CS(AR)
0B8	CS(BASE)
0BC	CS(LIMIT)
0C0	ES(AR)
0C4	ES(BASE)
0C8	ES(LIMIT)