vx32 Architecture Specification
DRAFT Version 0.1

Bryan Ford
Massachusetts Institute of Technology

December 5, 2006

1 Introduction

The vx32 architecture is a variant of the industry-standard (32-bit) x86 architecture, slightly modified in ways that are generally transparent to modern compiler-generated code, in order to make the architecture slightly more streamlined and amenable for use as a basis for portable, cross-platform virtual execution environments. The key differences between vx32 and x86-32 can be summarized as follows:

- Unlike x86, all vx32 instructions have fully-specified, completely deterministic behavior. This deterministic behavior guarantees that a program always computes the same result under any correct vx32 implementation (even if the program itself is buggy), and makes it easier to test the correctness of vx32 implementations.
- Only a subset of the x86-32 instruction set is available in vx32: in particular, no privileged instructions, no instructions relating to the x86's segmentation features, and no legacy binary-coded decimal arithmetic instructions. These reductions simplify vx32 implementations without affecting typical compiled code at all, since compiled code in general never uses these instructions.
- The legacy x87 floating-point unit (FPU) is omitted from vx32 in favor of the new SSE2 vector floating-point unit. This restriction ensures that all floating-point arithmetic, including transcendental operations, is fully deterministic across hardware platforms, and allows efficient vx32 emulation on non-x86 hardware platforms.
- Some redundant instruction encodings in x86-32 are not allowed in vx32, instead producing undefined opcode exceptions. Redundant or useless prefixes, for example, such as operand size prefixes for byte-size instructions, are not allowed in vx32. This restriction does not affect code generated by sensible compilers or assemblers, and reduces the diversity and lengths of instruction encodings that vx32 implementations must handle.

The vx32 architecture can be implemented efficiently on existing x86 processors via relatively simple and efficient instruction scanning and translation, at a performance cost typically less than 15%. Emulating vx32 efficiently on other hardware architectures requires more involved instruction translation, of course, but the slight changes vx32 makes with respect to x86-32 considerably lessen the difficulty of efficient cross-architecture translation.

2 Base Architecture

This section introduces the basic architectural features of vx32 and summarizes their differences from x86-32.

2.1 Registers

The vx32 programming model includes *only* the following registers:

- The eight 32-bit x86 general-purpose registers (GPRs): EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP.

- The 32-bit instruction pointer EIP.
- The 32-bit processor flags register EFLAGS (but only a subset of the flag bits are implemented, as defined below).
- The eight 128-bit XMM floating-point/vector registers.
- The 32-bit floating-point/vector control register MXCSR.

The vx32 programming model does *not* include the x86 segment registers CS, DS, ES, FS, GS, or SS, any of the x87 FPU or MMX registers, or any registers in the privileged portion of the x86-32 architecture.

2.2 Data Formats

The vx32 architecture supports all of the same data formats as x86-32, with the exceptions of packed or unpacked binary-coded decimal (BCD) integers and 80-bit extended-precision floating point values. The vx32 instruction set in particular supports 8-bit, 16-bit, and 32-bit integers in the eight GPRs, with 8-bit operands restricted to the AX/BX/CX/DX registers as usual; and 8-, 16-, 32-, 64-, and 128-bit integer values as well as 32-bit and 64-bit IEEE floating-point values in the XMM registers.

2.3 Addressing

All of the 32-bit addressing modes in the x86-32 architecture remain valid in vx32. 16-bit addressing is unavailable in vx32, however; x86-32 instructions that specify a 16-bit address size via the address size prefix (0x67) cause an undefined opcode exception in vx32.

2.4 Flags Register

The vx32 EFLAGS register includes all of the unprivileged x86-32 flags except for AF (Auxiliary Carry Flag), which is relevant only to BCD arithmetic. In particular, the vx32 EFLAGS consists of the direction control flag DF, and the five status flags CF, ZF, SF, PF, and OF. The vx32 EFLAGS register includes none of the system flags that normally can only be modified in privileged moded on the x86.

All unimplemented EFLAGS bits read as 0 in vx32, and unlike the x86-32, attempts to modify unimplemented bits using the POPF instruction cause a General Protection exception rather than just being ignored by the processor.

The behavior of various x86 instructions that affect the EFLAGS register are changed slightly in vx32, as specified in Section **??**, though only in ways that do not affect modern compiled code. In particular, vx32 instructions never leave EFLAG bits in an "undefined" state, since doing so would inherently make the architecture non-deterministic. Also, all arithmetic instructions that *ever* modify *any* of the status flags (CF, ZF, SF, PF, OF) are changed in vx32 so that they *always* set *all five* status flags deterministically. This change facilitates high-performance implementations of vx32 both in hardware, by reducing false EFLAGS dependencies across instructions that impede instruction-level parallelism, and in software, by making it unnecessary for instruction translators to track and simulate EFLAGS result bits resulting from several different instructions at a time.

2.5 Floating-Point Support

The vx32 architecture omits support for the legacy x87 floating-point unit (FPU), in favor of the newer SSE2 vector floating-point instructions. This significant omission is made for two reasons:

- The x87 FPU uses an 80-bit "extended-precision" floating point format to hold intermediate arithmetic results, instead of the IEEE-standard 32-bit or 64-bit floating-point formats that are generally supported on other processor architectures. If the vx32 included x87 support, the x87's 80-bit registers and floating-point arithmetic would have to be emulated entirely in software on essentially all non-x86 hardware platforms, substantially reducing vx32's potential usefulness as a cross-hardware-platform virtual environment.

General-Purpose Integer Instructions	
Data Transfer	MOV, MOVNTI, CMOV*cc*, PUSH, POP, XCHG
Arithmetic	ADD, SUB, ADC, SBB, NEG, MUL, IMUL, DIV, IDIV, INC, DEC
Data Conversion	CBW, CWD, CWDE, CDQ, MOVSX, MOVZX
Shift/Rotate	SHL, SHR, SHLD, SHRD, SAL, SAR, ROL, ROR, RCL, RCR
Compare/Test	CMP, TEST, BSF, BSR, BT, BTS, BTR, BTC, SET*cc*
Logical	AND, OR, XOR, NOT
String	CMPS, SCAS, MOVS, LODS, STOS
Control Transfer	JMP, J*cc*, LOOP*cc*, CALL, RET
Flags	PUSHF, POPF, CLC, CMC, STC, CLD, STD
Cache Control	PREFETCH, CLFLUSH, LFENCE, SFENCE, MFENCE
Miscellaneous	BSWAP, LEA, ENTER, LEAVE, NOP

Table 1: The vx32 Base Instruction Set

- Due to their basic mathematical properties, most of the legacy x87 FPU's transcendental instructions, such as sine, cosine, and exponentiation, are fundamentally impractical to make fully deterministic without actually specifying a particular implementation. Since any such a prescribed implementation would inevitably differ from those of x86 hardware platforms, thus yielding (slightly) different results in different situations, a vx32 environment would always have to emulate these instructions in software to ensure full determinism, even when running on x86-based hardware platforms that have built-in x87 support.

By restricting software to the newer, fully deterministic and IEEE-compliant SSE2 instructions, most x86 software that uses floating-point can be ported to vx32 with no more than a recompile using the appropriate compiler options, and non-floating-point software may not even need to be recompiled. Legacy code that really needs 80-bit extended precision x87 floating-point can still use a software x87 FPU emulator running *inside* the vx32 environment. This solution provides backward compatibility without sacrificing deterministic execution, because the same emulation code will be used, and will thus produce the same results, under any correct vx32 environment.

3 Instruction Set

The vx32 instruction set is organized into a set of *base instructions* and two *numeric extensions* that include successively larger sets of instructions. The base instruction set roughly corresponds to the x86's unprivileged integer instruction set, while the numeric extensions correspond to the x86's SSE/SSE2 instructions. The two numeric extensions are organized not according to the historical order in which the various SSE instructions were introduced in x86 processors, but rather logically according to their functional purpose. Separating the numeric extensions from the base instruction set in this way facilitates the use of well-defined vx32 subsets in environments for which the full computational power of the x86 architecture is unnecessary and may impose an unreasonable implementation cost.

3.1 Base Instruction Set

The vx32 base instruction set includes all of, and only, those instructions listed in Table **??**. The instructions behave and are encoded as described in standard x86 references, except as specified elsewhere in this document.

3.2 Floating-Point Extension

The first vx32 numeric extension adds support for scalar 32-bit and 64-bit IEEE floating-point arithmetic, providing the basic floating-point primitives that most high-level languages and compilers can take advantage of directly. The vx32 floating-point extension includes the base instructions listed in Table **??**, plus the instructions listed in Table **??**.

Scalar Single-Precision Floating-Point	
Data Transfer	MOVD, MOVSS
Data Conversion	CVTSI2SS, CVTSS2SI, CVTTSS2SI, CVTSS2SD, CVTSD2SS
Arithmetic	ADDSS, SUBSS, MULSS, DIVSS, SQRTSS
Comparison	CMPSS, MAXSS, MINSS, COMISS, UCOMISS
Scalar Double-Precision Floating-Point	
Data Transfer	MOVQ, MOVSD
Data Conversion	CVTSI2SD, CVTSD2SI, CVTTSD2SI, CVTSS2SD, CVTSD2SS
Arithmetic	ADDSD, SUBSD, MULSD, DIVSD, SQRTSD
Comparison	CMPSD, MAXSD, MINSD, COMISD, UCOMISD
Floating-Point Control	
Control	STMXCSR, LDMXCSR

Table 2: The vx32 Floating-Point Extension

3.3 Vector Processing Extension

The second numeric extension builds on the floating-point extension, and further adds support for packed integer and floating-point arithmetic. The vx32 vector processing extension includes all of the instructions listed in Tables **??** and **??**, plus the additional instructions listed in Table **??**.

3.4 Omitted Instructions

For informational purposes, this section lists the traditional x86-32 instructions that are left out of the vx32 instruction set, and the rationale for omitting them.

3.4.1 Omitted Integer Instructions

- **PUSHA, POPA:** These instructions are rarely used and never generated by compilers, because it is generally faster, more flexible, and nearly as easy to save only the registers that need to be saved. And because they perform many memory accesses in one instruction, it is nontrivial to specify precisely their exact exception semantics in corner cases when an exception may occur partway through.
- **XLAT:** This translation table lookup instruction is never generated by compilers and has historically never gotten much use because its operation is far too specialized.
- **AAA, AAD, AAM, AAS, DAA, DAS:** These Binary-Coded Decimal (BCD) arithmetic instructions are never used in modern x86 code, because it has proven far more efficient just to do arithmetic in binary and convert to and from decimal as necessary.
- **XADD, CMPXCHG, and CMPXCHG8B:** These instructions only exist to provide atomic synchronization primitives in multiprocessor systems. Although vx32 environments can run on multiprocessor systems, any particular vx32 environment must be single-threaded, running on only one processor at a time, because a multi-threaded vx32 environment would inherently have to be non-deterministic.
- **INT, INTO, BOUND:** XXX ???
- **LAHF, SAHF:** XXX ???
- **CPUID:** XXX ??? Because the deterministic model implies that the environment must not be able to know what type of physical CPU it is running on.
- **SYSENTER, SYSEXIT, SYSCALL, SYSRET:** XXX ??? Because we have a quite different notion of system calls and interaction between parent and child environments.

	Integer Vector Arithmetic
Data Transfer	MOVDQA, MOVDQU, MOVNTDQ, MASKMOVDQU, PMOVMSKB
Data Reordering	PACKSSDW, PACKSSWB, PACKUSWB, PEXTRW, PINSRW, PSHUFD, PSHUFHW, PSHUFLW PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ, PUNPCKHQDQ, PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ, PUNPCKLQDQ,
Arithmetic	PADDB, PADDW, PADDD, PADDQ, PADDSB, PADDSW, PADDUSB, PADDUSW, PSUBB, PSUBW, PSUBD, PSUBQ, PSUBSB, PSUBSW, PSUBUSB, PSUBUSW, PMULHW, PMULLW, PMULHUW, PMULUDQ, PMADDWD, PAVGB, PAVGW, PSADBW
Shift	PSLLW, PSLLD, PSLLQ, PSLLDQ, PSRLW, PSRLD, PSRLQ, PSRLDQ, PSRAW, PSRAD
Comparison	PCMPEQB, PCMPEQW, PCMPEQD, PCMPGTB, PCMPGTW, PCMPGTD, PMAXUB, PMINUB, PMAXSW, PMINSW
Logical	PAND, PANDN, POR, PXOR
	Packed Single-Precision Floating-Point
Data Transfer	MOVAPS, MOVUPS, MOVHPS, MOVLPS, MOVHLPS, MOVLHPS, MOVNTPS, MOVMSKPS
Data Conversion	CVTDQ2PS, CVTPS2DQ, CVTTPS2DQ, CVTPS2PD, CVTPD2PS
Data Reordering	UNPCKHPS, UNPCKLPS, SHUFPS
Arithmetic	ADDPS, SUBPS, MULPS, DIVPS, SQRTPS
Comparison	CMPPS, MAXPS, MINPS
Logical	ANDPS, ANDNPS, ORPS, XORPS
	Packed Double-Precision Floating-Point
Data Transfer	MOVAPD, MOVUPD, MOVHPD, MOVLPD, MOVNTPD, MOVMSKPD
Data Conversion	CVTDQ2PD. CVTPD2DQ, CVTTPD2DQ, CVTPS2PD, CVTPD2PS
Data Reordering	UNPCKHPD, UNPCKLPD, SHUFPD
Arithmetic	ADDPD, SUBPD, MULPD, DIVPD, SQRTPD
Comparison	CMPPD, MAXPD, MINPD
Logical	ANDPD, ANDNPD, ORPD, XORPD

Table 3: The vx32 Vector Processing Extension

3.4.2 Omitted Numeric Instructions

- **RCPSS, RCPPS, RSQRTSS, RSQRTPS:** These SSE2 instructions are defined in the x86 architecture to produce only approximate rather than exact results, which implies that they can produce different values across different implementations and thus lack determinism. (XXX just make them compute the obvious exact results under vx32?)

4 Instruction Behavior Differences

This section describes differences in the behaviors of certain instructions between vx32 and x86-32.

4.1 16-bit Shift and Rotate Instructions

Shift and rotate instructions with 16-bit operands in vx32 use only the low four bits of the shift count specified in the immediate operand or the CL register. This behavior contrasts with x86-32, which uses the low five bits of the shift count but leaves the result undefined if the masked shift count is greater than 16.

This difference can only affect correct x86-32 code if the code performs a 16-bit shift with a count of exactly 16: such an instruction is effectively a 0-bit shift in vx32, but acts as a 16-bit shift in x86-32. This subtle difference should not affect reasonable x86-32 code, which is more likely to use 32-bit operands in the first place. 32-bit compilers for languages such as C, C++, or Java cannot use 16-bit x86 shift instructions even on 16-bit variables because of the implicit promotions to 32-bit `int` types that the language standards require.

4.2 Segment Register Access Instructions

The variants of the MOV, PUSH, and POP instructions that access segment registers in the x86 architecture are unavailable and cause illegal opcode exceptions in vx32.

4.3 Instructions Supporting 32-bit Operands Only

The 16-bit forms of the PUSH and POP instructions are unavailable and cause illegal opcode exceptions in vx32. Sensible 32-bit x86 code always keeps the stack pointer aligned to a 32-bit boundary, rendering the 16-bit PUSH and POP instructions obsolete.

4.4 ENTER instruction

The second immediate operand of the ENTER instruction (the nesting level) must be 0 on vx32, otherwise an illegal opcode exception results. The corner-case semantics of this instruction are ill-defined (e.g., what happens when the nesting level specified is greater than 31, or when the processor takes an exception while pushing frame pointers), and modern compilers don't generally use the nesting facility anyay.

4.5 Memory Access Ordering Instructions

The memory access ordering instructions LFENCE, SFENCE, MFENCE are allowed but are architectural no-ops in vx32. Multiprocessor write ordering issues never come into play within vx32 environments, because a vx32 environment must be single-threaded in order to be deterministic.

5 Status Flags Behavior

Table **??** summarizes the effects on the five status flag bits in EFLAGS (OF, SF, ZF, PF, and CF) by all vx32 instructions that affect the status flags as a side-effect of their execution. The list does not include instructions such as CLC whose primary function is to modify a particular flag; such instructions behave exactly as in x86-32.

	OF	SF	ZF	PF	CF
ADC	M	M	M	M	TM
ADD	M	M	M	M	M
AND	0	M	M	M	0
BSF,BSR	0*	0*	M	0*	0*
BT,BTS,BTR,BTC	0*	0*	0*	0*	M
CMP	M	M	M	M	M
CMPS	M	M	M	M	M
CMPXCHG	M	M	M	M	M
COMISD, COMISS	0	0	M	M	M
DEC	M	M	M	M	M*
DIV	0*	0*	0*	0*	0*
IDIV	0*	0*	0*	0*	0*
IMUL	M	0*	0*	0*	M
INC	M	M	M	M	M*
MUL	M	0*	0*	0*	M
NEG	M	M	M	M	M
OR	0	M	M	M	0
RCL,RCR	M*	M*	M*	M*	TM*
ROL,ROR	M*	M*	M*	M*	M*
SAL,SAR,SHR	M*	M*	M*	M*	M*
SBB	M	M	M	M	TM
SCAS	M	M	M	M	M
SHLD,SHRD	M*	M*	M*	M*	M*
SUB	M	M	M	M	M
TEST	0	M	M	M	0
UCOMISD, UCOMISS	0	0	M	M	M
XADD	M	M	M	M	M
XOR	0	M	M	M	0

M: instruction modifies flag according to result.
T: instruction tests flag.
0: instruction unconditionally sets flag zero.
*: behavior differs from x86.

Table 4: Side-effects of vx32 instructions on status flags. M: instruction modifies flag according to result. T: instruction tests flag. 0: instruction sets flag bit t0 zero. *: behavior differs from x86.

5.1 Instructions that Leave Status Flags Undefined in x86

The x86 architecture leaves various status flags in the EFLAGS register in undefined states after executing certain instructions: IDIV leaves all of the status flags undefined, for example. The vx32 architecture defines specific values for all such results, in order to make execution fully deterministic across all vx32 implementations. With a few exceptions described below, flag bit results that the x86 architecture leaves undefined are forced to 0 in vx32.

The affected instructions are:

- **Multiply:** MUL and IMUL set SF, ZF, and PF to zero in vx32. (XXX should be set according to result?)
- **Divide:** DIV and IDIV set all status flags to zero in vx32. (XXX should be set according to result?)
- **Bit-scan:** BSF and BSR set all flags except ZF to zero in vx32.
- **Bit test/modify:** BT, BTS, BTR, and BTC set all flags except CF to zero in vx32.

5.2 Instructions that Partially Set Status Flags in x86

A few instructions are architecturally defined in x86-32 to affect certain flags while leaving others unmodified, or affect the status flags only under some conditions. While originally intended to facilitate certain code optimizations, this behavior creates false dependencies between instructions that interfere with the ability of modern superscalar processors to extract instruction-level parallelism from the code. For this reason modern 32-bit compilers generally avoid using these instructions when optimizing for performance, and in any case do not generally write code that depends on flag bits "tunneling through" instructions that only partially modify the status flags. Instructions that partially modify the status flags are also difficult to implement efficiently in software instruction translators, because of the need to track status flags that may have been generated by multiple preceding instructions. For these reasons, vx32 modifies the following instructions so that they *always* set *all* of the status flags:

- **Increment/Decrement:** The INC *rm* and DEC *rm* instructions in vx32 are exactly equivalent to ADD *rm*,1 and SUB *rm*,1, respectively, in contrast with the x86 architecture in which INC and DEC leave the carry flag unmodified.
- **Shift:** The shift instructions SAL, SAR, SHR, SHLD, and SHRD always modify all status flags according to the result, including in the case of a shift count of zero in which the status flags would remain unmodified on the x86. SAL and SHLD on vx32 always leave the OF flag containing the XOR of the CF with the most-significant bit of the result, regardless of the shift count, unlike on x86 in which OF is defined only for a shift count of 1. Similarly, SAR leaves the OF cleared, SHR leaves the OF containing a copy of the ... containing SHRD on vx32 als ... XXX ...and CF for shift count of 0?
- **Rotate:** The rotate instructions ROL, ROR, RCL, and RCR always modify all status flags according to the result, in contrast with x86 behavior in which only the CF and OF flags are ever modified, and only for non-zero shift counts. ROL and RCL on vx32 always leave OF containing the XOR of the final CF with the most-significant bit of the result, regardless of the shift count, while ROR and RCR always leave OF containing the XOR of the two most-significant bits of the result.

6 Instruction Encodings

no 16-bit code or 16-bit addressing...

disallowed redundant prefixes...

7 Implementing VX32 on the x86

7.1 INC and DEC instructions

vx32 instruction	x86 equivalent code	Comments
INC ea	ADD ea,1	Add, setting all status flags
DEC ea	SUB ea,1	Subtract, setting all status flags

7.2 Shift instructions

7.2.1 SAL and SHL

vx32 instruction	x86 equivalent code	Comments
SxL ea,1	SxL ea,1	Shift, set all status flags
SxL ea,$1+n$	SxL ea,n	Shift all but one bit
	SxL ea,1	Shift last bit, set status flags
SxL ea,0	CMP ea,0	Set status flags except OF
	RCR ea,1	Set OF correctly
	RCL ea,1	
SxL ea,CL	CMP ea,0	Set flags except OF for CL=0
	SxL ea,CL	Shift, set flags for CL>0
	RCR ea,1	Set OF correctly for CL>1
	RCL ea,1	

SAR and SHR:

vx32 instruction	x86 equivalent code	Comments
SxR ea,1	SxR ea,1	Shift, set all status flags
SxR ea,$1+n$	SxR ea,n	Shift all but one bit
	SxR ea,1	Shift last bit, set status flags
SxR ea,0	CMP ea,0	Set status flags except OF
	RCL ea,1	Set OF correctly
	RCR ea,1	
SxR ea,CL	CMP ea,0	Set flags except OF for CL=0
	SxR ea,CL	Shift, set flags for CL>0
	RCL ea,1	Set OF correctly for CL>1
	RCR ea,1	

SHLD:

vx32 instruction	x86 equivalent code	Comments
SHLD ea,reg,1	SHLD ea,reg,1	Shift, set all status flags
SHLD ea,reg,$1+n$	SHLD ea,reg,n	Shift all but one bit
	SHLD ea,reg,1	Shift last bit, set status
SHLD ea,reg,0	CMP ea,0	Set status flags except OF
	RCR ea,1	Set OF correctly
	RCL ea,1	
SHLD ea,reg,CL	CMP ea,0	Set flags for CL=0
	SHLD ea,reg,CL	Shift, set flags for CL>0
	RCR ea,1	Set OF correctly for CL>1
	RCL ea,1	

SHRD:

vx32 instruction	x86 equivalent code	Comments
SHRD ea,reg,1	SHRD ea,reg,1	Shift, set all status flags
SHRD ea,reg,$1+n$	SHRD ea,reg,n	Shift all but one bit
	SHRD ea,reg,1	Shift last bit, set status
SHRD ea,reg,0	CMP ea,0	Set status flags except OF
	RCL ea,1	Set OF correctly
	RCR ea,1	
SHRD ea,reg,CL	CMP ea,0	Set flags for CL=0
	SHRD ea,reg,CL	Shift, set flags for CL>0
	RCL ea,1	Set OF correctly for CL>1
	RCR ea,1	

7.3 Shift Instructions

Problems:

- On x86, all flags are unmodified if shift count is zero.
- On x86, resulting OF is undefined if shift count is greater than zero.
- On x86, result of SHLD/SHRD is undefined if shift count is greater than operand size, which can only happen on 16-bit shifts.

Solution:

- For immediate shifts, first mask the shift count against (operand size - 1): e.g., mask it to 4 bits for 16-bit shifts and 5 for 32-bit shifts. Then:
  - If result flags are not needed, emit a shift instruction with the masked shift count.
  - Otherwise, if resulting shift count is zero, rewrite the shift instruction to a CMP that compares the destination with itself.
  - If resulting shift count is one, leave the instruction as-is.
  - If resulting shift count n is greater than one, emit a shift instruction with a count of $n-1$ followed by a 1-bit shift instruction.
- For 32-bit variable shifts, first emit the original shift/rotate instruction. Then, if the instruction's result flags are needed, emit a 1-bit rotate through carry in the opposite direction as the original instruction, followed by a 1-bit rotate through carry in the appropriate direction.
- For 16-bit variable shifts, save the ECX register in memory, mask ECX to the low 4 bits, emit the appropriate variable shift instruction, and restore the ECX register. Then, if the instruction's result flags are needed, emit a pair of 1-bit rotates as for 32-bit variable shifts.

8 Emulating VX32 on Other Architectures

Other architectures of interest that are currently (still) commercially viable: PowerPC, used in the modern Apple Macintosh, and MIPS and ARM, used extensively in embedded computing environments for PDAs, cell phones, etc.

8.1 PowerPC

Appears quite practical overall. A few likely minor difficulties:

- Alignment checking: PowerPC implementations are not architecturally required to check for and take exceptions on attempts to perform unaligned memory accesses.

- Floating-point registers: PCC has separate register files for scalar versus vector FP arithmetic, whereas SSE2-based x86 floating-point arithmetic uses a single register file. Also, PPC expands single-precision scalar floating-point values into double-precision format within its scalar FP registers. Finally, the PPC AltiVec vector processing unit supports only single-precision FP, not double precision as on x86.
- Missing operations: AltiVec does not provide vector square root or divide instructions, as x86 SSE2 does. (But the AltiVec manual shows how to emulate them, apparently with exact IEEE-compliant rounding behavior.)

8.2 MIPS

8.3 ARM