GAS IA-32 Assembly Language Programming
Floating Point Unit Instructions

Tony Richardson
University of Evansville
April 19, 2007

The following table contains a partial list of IA-32 assembly floating point mnemonics. An *S* in the mnemonic represents a size specification letter and should be used when an operand is a memory reference. If the reference is to a floating point type the suffix should be s (single, 4 bytes), l (double, 8 bytes) or t (long double, 10 bytes). If the reference is to an integer type the suffix should be s (short, 2 bytes), l (int, 4 bytes) or q (long int, 8 bytes). If the operands are only FPU registers then a suffix should *not* be used. **fld** and **fstp** are the only floating point related instructions that allow the t suffix. **fild** and **fistp** are the only integer related instructions that allow a q suffix.

Although the eight FPU stack registers are referred to as ST0 through ST7 in the documentation below in **gas** use the notation %st(0) through %st(7) to refer to the registers.

In the operand columns an **r** represents an FPU register and an **m** a memory location. A destination operand appears on the same line as a corresponding source operand.

This is only a partial list of FPU mnemonics. The table was extracted from the NASM documentation and converted to AT&T syntax. I recommend the NASM documentation (as well as the Intel documentation) as a complete source of documentation for IA-32 assembly language programming.

Mnemonic	**Op 1 (SRC)**	**Op 2 (DEST)**	**Description**
f2xm1			**Calculate 2**X-1** F2XM1 raises 2 to the power of ST0, subtracts one, and stores the result back into ST0. The initial contents of ST0 must be a number in the range -1.0 to +1.0.
fabs			**Absolute Value** FABS computes the absolute value of ST0, by clearing the sign bit, and stores the result back in ST0.
fadd*S*	r/m r ST0	 ST0 r	**Addition** FADD, given one operand, adds the operand to ST0 and stores the result back in ST0. With two operands the results is stored in the destination (second) register. To add an integer value to ST0, use the FIADD instruction
faddp*S*	ST0	r	**Addition** FADDP performs the same function as FADD, but pops the register stack after storing the result.
fchs			**Change Sign** FCHS negates the number in ST0, by inverting the sign bit: negative numbers become positive, and vice versa.
fcom*S* fcomp*S* fucom*S* fucomp*S*	m/r r	 ST0	**Compare** FCOM compares ST0 with the given operand, and sets the FPU flags accordingly. ST0 is treated as the left-hand side of the comparison, so that the carry flag is set (for a `less-than' result) if ST0 is less than the given operand.  FCOMP does the same as FCOM, but pops the register stack afterwards.  The FUCOM instructions differ from the FCOM instructions only in the way they handle quiet NaNs: FUCOM will handle them silently and set the condition code flags to an `unordered' result, whereas FCOM will generate an exception.

fcompp fucompp			**Compare** FCOMP does the same as FCOM, but pops the register stack afterwards. FCOMPP compares ST0 with ST1 and then pops the register stack twice.
fcomi fcomip fucomi fucomip	r		**Compare** FCOMI and FCOMIP work like the corresponding forms of FCOM and FCOMP, but write their results directly to the CPU flags register rather than the FPU status word, so they can be immediately followed by conditional jump or conditional move instructions.
fcos			**Cosine** FCOS computes the cosine of ST0 (in radians), and stores the result in ST0. The absolute value of ST0 must be less than 2**63. See also FSINCOS.
fdecstp			**Decrement Floating-Point Stack Pointer** FDECSTP decrements the `top' field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the contents of ST7 had been pushed on the stack. See also FINCSTP.
fdiv*S* fdivr*S*	m/r ST0 r	 r ST0	**Division** FDIV, given one operand, divides ST0 by the operand and stores the result back in ST0. The two operand form always divides ST0 by the other operand and stores the result in the second operand. FDIVR does the same thing, but does the division the other way round. The one operand form divides the operand by ST0 and stores the result in ST0. The two operand from always divides the other operand by ST0 and stores the result in the second operand. For FP/Integer divisions, see FIDIV
fdivp fdivrp	r ST0	 r	**Division** FDIVP operates like FDIV, but pops the register stack once it has finished. FDIVRP operates like FDIVR, but pops the register stack once it has finished.
ffree ffreep	r		**Free Register** FFREE marks the given register as being empty. FFREEP marks the given register as being empty, and then pops the register stack.
fiadd*S*	m		**Floating-Point/Integer Addition** FIADD adds the 16-bit or 32-bit integer stored in the given memory location to ST0, storing the result in ST0.
ficom*S* ficomp*S*	m		**Floating-Point/Integer Compare** FICOM compares ST0 with the 16-bit or 32-bit integer stored in the given memory location, and sets the FPU flags accordingly. FICOMP does the same, but pops the register stack afterwards.
fidiv*S* fidivr*S*	m		**Floating-Point/Integer Division** FIDIV divides ST0 by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in ST0. FIDIVR does the division the other way up: it divides the integer by ST0, but still stores the result in ST0.
fild*S* fist*S* fistp*S*	m		**Floating-Point/Integer Conversion** FILD loads an integer out of a memory location, converts it to a real, and pushes it on the FPU register stack. FIST converts ST0 to an integer and stores that in memory; FISTP does the same as FIST, but pops the register stack afterwards. (FIST does not allow the q suffix.)
fimul*S*	m		**Floating-Point/Integer Multiplication** FIMUL multiplies ST0 by the 16-bit or 32-bit integer stored in the given memory location, and stores the result in ST0.

fincstp			**Increment Floating-Point Stack Pointer** FINCSTP increments the "top" field in the floating-point status word. This has the effect of rotating the FPU register stack by one, as if the register stack had been popped; however, unlike the popping of the stack performed by many FPU instructions, it does not flag the new ST7 (previously ST0) as empty. See also FDECSTP.
finit fninit			**Initialise Floating-Point Unit** FINIT initialises the FPU to its default state. It flags all registers as empty, without actually change their values, clears the top of stack pointer. FNINIT does the same, without first waiting for pending exceptions to clear.
fisub*S* fisubr*S*	m		**Floating-Point/Integer Subtraction** FISUB subtracts the 16-bit or 32-bit integer stored in the given memory location from ST0, and stores the result in ST0. FISUBR does the subtraction the other way round, i.e. it subtracts ST0 from the given integer, but still stores the result in ST0.
fld*S*	m/r		**Load** FLD loads a floating-point value out of the given register or memory location, and pushes it on the FPU register stack.
fld1 fldl2e fldl2t fldlg2 fldln2 fldpi fldz			**Load Constants** These instructions push specific standard constants on the FPU register stack. *Instruction* — *Constant pushed* FLD1 — 1 FLDL2E — base-2 logarithm of e FLDL2T — base-2 log of 10 FLDLG2 — base-10 log of 2 FLDLN2 — base-e log of 2 FLDPI — pi FLDZ — zero
fmul*S*	m/r r ST0	 ST0 r	**Multiply** FMUL, given one operand, multiplies ST0 by the given operand, and stores the result in ST0. With two operands it stores the result in the second operand.
fmulp	ST0	r	**Multiply** FMULP performs the same operation as FMUL, and then pops the register stack.
fnop			**No Operation** FNOP does nothing.
fpatan fptan			**Arctangent and Tangent** FPATAN computes the arctangent, in radians, of the result of dividing ST1 by ST0, stores the result in ST1, and pops the register stack. It works like the C atan2 function, in that changing the sign of both ST0 and ST1 changes the output value by pi (so it performs true rectangular-to-polar coordinate conversion, with ST1 being the Y coordinate and ST0 being the X coordinate, not merely an arctangent). FPTAN computes the tangent of the value in ST0 (in radians), and stores the result in ST0. A value of 1.0 is then pushed on the stack, shifting the tangent to ST1. (This allows the cotangent to be easily computed using FDIV.) The absolute value of ST0 must be less than 2**63.
fprem fprem1			**Partial Remainder** These instructions both produce the remainder obtained by dividing ST0 by ST1. This is calculated, notionally, by dividing ST0 by ST1, rounding the result to an integer, multiplying by ST1 again, and computing the value which would need to be added back on to the result to get back to the original value in ST0.

			The two instructions differ in the way the notional round-to-integer operation is performed. FPREM does it by rounding towards zero, so that the remainder it returns always has the same sign as the original value in ST0; FPREM1 does it by rounding to the nearest integer, so that the remainder always has at most half the magnitude of ST1. Both instructions calculate partial remainders, meaning that they may not manage to provide the final result, but might leave intermediate results in ST0 instead. If this happens, they will set the C2 flag in the FPU status word; therefore, to calculate a remainder, you should repeatedly execute FPREM or FPREM1 until C2 becomes clear.
frndint			**Round to Integer** FRNDINT rounds the contents of ST0 to an integer, according to the current rounding mode set in the FPU control word, and stores the result back in ST0.
fsave fnsave frstor			**Save/Restore Floating-Point State** FSAVE saves the entire floating-point unit state, including all the information saved by FSTENV plus the contents of all the registers, to a 94 or 108 byte area of memory (depending on the CPU mode). FRSTOR restores the floating-point state from the same area of memory. FSAVE empties the FPU stack. FNSAVE does the same as FSAVE, without first waiting for pending floating-point exceptions to clear.
fscale			**Scale Value by Power of Two** FSCALE scales a number by a power of two: it rounds ST1 towards zero to obtain an integer, then multiplies ST0 by two to the power of that integer, and stores the result in ST0.
fsin fsincos			**Sine and Cosine** FSIN calculates the sine of ST0 (in radians) and stores the result in ST0. FSINCOS does the same, but then pushes the cosine of the same value on the register stack, so that the sine ends up in ST1 and the cosine in ST0. FSINCOS is faster than executing FSIN and FCOS (see section B.4.74) in succession. The absolute value of ST0 must be less than 2**63.
fsqrt			**Square Root** FSQRT calculates the square root of ST0 and stores the result in ST0.
fst*S* fstp*S*	m/r		**Store** FST stores the value in ST0 into the given memory location or other FPU register. FSTP does the same, but then pops the register stack. (FST does not allow the **t** suffix.)
fstcw fnstcw			**Store Control Word** FSTCW stores the FPU control word (governing things like the rounding mode, the precision, and the exception masks) into a 2-byte memory area. See also FLDCW. FNSTCW does the same thing as FSTCW, without first waiting for pending floating-point exceptions to clear.
fstenv fnstenv	m		**Store Environment** FSTENV stores the FPU operating environment (control word, status word, tag word, instruction pointer, data pointer and last opcode) into memory. The memory area is 14 or 28 bytes long, depending on the CPU mode at the time. See also FLDENV. FNSTENV does the same thing as FSTENV, without first waiting for pending floating-point exceptions to clear.
fstsw	m		**Store Status Word**

fnstsw	AX		FSTSW stores the FPU status word into AX or into a 2-byte memory area. FNSTSW does the same thing as FSTSW, without first waiting for pending floating-point exceptions to clear.
fsub*S* fsubr*S*	m/r r ST0	 ST0 r	**Subtract** FSUB, given one operand, subtracts the operand from ST0 and stores the result back in ST0. With two operands it subtracts the other operand from ST0 and stores the result in the second operand. FSUBR does the same thing, but does the subtraction the other way round. So, with one operand, it subtracts ST0 from the operand and stores the result in ST0. With two operands, it subtracts ST0 from the other operand and stores the result in the second operand.
ftst			**Test ST0 Against 0** FTST compares ST0 with zero and sets the FPU flags accordingly. ST0 is treated as the left-hand side of the comparison, so that a `less-than' result is generated if ST0 is negative.
fxch	 r ST0 r	 r ST0	**Exchange** FXCH exchanges ST0 with a given FPU register. The no-operand form exchanges ST0 with ST1.
fxrstor	m		**Restore State** The FXRSTOR instruction reloads the FPU, MMX and SSE state (environment and registers), from the 512 byte memory area defined by the source operand. This data should have been written by a previous FXSAVE.
fxsave	m		**Store State** The FXSAVE instruction writes the current FPU, MMX and SSE technology states (environment and registers), to the 512 byte memory area defined by the destination operand. It does this without checking for pending unmasked floating-point exceptions (similar to the operation of FNSAVE). Unlike the FSAVE/FNSAVE instructions, the processor retains the contents of the FPU, MMX and SSE state in the processor after the state has been saved. This instruction has been optimised to maximize floating-point save performance.
fxtract			**Extract Exponent and Significand** FXTRACT separates the number in ST0 into its exponent and significand (mantissa), stores the exponent back into ST0, and then pushes the significand on the register stack (so that the significand ends up in ST0, and the exponent in ST1).
fyl2x fyl2xp1			**Compute Y times Log2(X) or Log2(X+1)** FYL2X multiplies ST1 by the base-2 logarithm of ST0, stores the result in ST1, and pops the register stack (so that the result ends up in ST0). ST0 must be non-zero and positive. FYL2XP1 works the same way, but replacing the base-2 log of ST0 with that of ST0 plus one. This time, ST0 must have magnitude no greater than 1 minus half the square root of two.