CS:APP2e Web Aside ASM:X87:
X87-Based Support for Floating Point[*]

Randal E. Bryant
David R. O'Hallaron

June 5, 2012

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Perspective, Second Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2011. In this document, all references beginning with "CS:APP2e " are to this book. More information about the book is available at* `csapp.cs.cmu.edu`.

This document is being made available to the public, subject to copyright provisions. You are free to copy and distribute it, but you should not use any of this material without attribution.

1 Introduction

The *floating-point architecture* for a processor consists of the different aspects that affect how programs operating on floating-point data are mapped onto the machine, including:

- How floating-point values are stored and accessed. This is typically via some form of registers.
- The instructions that operate on floating-point data.
- The conventions used for passing floating-point values as arguments to functions, and for returning them as results.

In this document, we will describe the floating-point architecture for x86 processors known as *x87*.

The set of instructions for manipulating floating-point values is one of the least elegant features of the historical x86 architecture. In the original Intel machines, floating point was performed by a separate *coprocessor*, a unit with its own registers and processing capabilities that executes a subset of the instructions. This coprocessor was implemented as a separate chip named the 8087, 80287, and i387, to accompany the processor chips 8086, 80286, and i386, respectively. During these product generations, chip capacity was insufficient to include both the main processor and the floating-point coprocessor on a single chip. In addition, lower-budget machines would omit floating-point hardware and simply perform the floating-point operations (very slowly!) in software. Since the i486, floating point has been included as part of the IA32 CPU chip. The legacy 8087 defines a set of instructions and a storage model for implementing floating-point code, often referred to as "x87," much as "x86" refers to the evolutionary processor architecture that started with the 8086. We will use the term "x87 instructions" in this document.

The original 8087 coprocessor was introduced to great acclaim in 1980. It was the first single-chip floating-point unit (FPU), and the first implementation of what became the IEEE 754 floating-point standard. Operating as a coprocessor, the FPU would take over the execution of floating-point instructions after they were fetched by the main processor. There was minimal connection between the FPU and the main processor. Communicating data from one processor to the other required the sending processor to write to memory and the receiving one to read it. Artifacts of that design remain in the x87 floating-point instruction set today. In addition, the compiler technology of 1980 was much less sophisticated than it is today. Many features of x87 make it a difficult target for optimizing compilers.

With the introduction of SSE2 in the Pentium 4 (2000), it has become possible to implement single and double-precision floating-point arithmetic using SSE instructions. These provide a much better target for optimizing compilers (see Web Aside ASM:SSE), and so slowly the use of the x87 floating-point architecture is being phased out of x86 code. Still, x87 instructions are the default for GCC when generating IA32 floating-point code. They are also the only way to implement 80-bit extended-precision floating-point operations, such as for C data type `long double`. In this document, we consider only IA32 code. All x87 instructions can be used with x86-64 code, as well, but the conventions for passing function arguments and returning function values in x86-64 code are based on the SSE floating-point architecture.

2 Floating-Point Registers

X87 has eight floating-point registers, but unlike normal registers, these are treated as a shallow stack. The registers are identified as `%st(0)`, `%st(1)`, and so on, up to `%st(7)`, with `%st(0)` being the top of the stack. When more than eight values are pushed onto the stack, the ones at the bottom simply disappear.

Rather than directly indexing the registers, most of the arithmetic instructions pop their source operands from the stack, compute a result, and then push the result onto the stack. Stack architectures were considered a clever idea in the 1970s, since they provide a simple mechanism for evaluating arithmetic instructions, and they allow a very dense coding of the instructions. With advances in compiler technology and with the memory required to encode instructions no longer considered a critical resource, these properties are no longer important. Compiler writers would be much happier with a conventional set of floating-point registers, such as is available with SSE.

Aside: Other stack-based languages.
Stack-based interpreters are still commonly used as an intermediate representation between a high-level language and its mapping onto an actual machine. Other examples of stack-based evaluators include Java byte code, the intermediate format generated by Java compilers, and the PostScript page formatting language. **End Aside.**

Having the floating-point registers organized as a bounded stack makes it difficult for compilers to use these

Instruction	Effect
`load` *S*	Push value at *S* onto stack
`storep` *D*	Pop top stack element and store at *D*
`neg`	Negate top stack element
`addp`	Pop top two stack elements; Push their sum
`subp`	Pop top two stack elements; Push their difference
`multp`	Pop top two stack elements; Push their product
`divp`	Pop top two stack elements; Push their ratio

Figure 1: **Hypothetical stack instruction set.** These instructions are used to illustrate stack-based expression evaluation

registers for storing the local variables of a procedure that calls other procedures. For storing local integer variables, we have seen that some of the general purpose registers can be designated as callee saved and hence be used to hold local variables across a procedure call. Such a designation is not possible for an x87 register, since its identity changes as values are pushed onto and popped from the stack. For example, a push operation causes the value in `%st(0)` to now be in `%st(1)`.

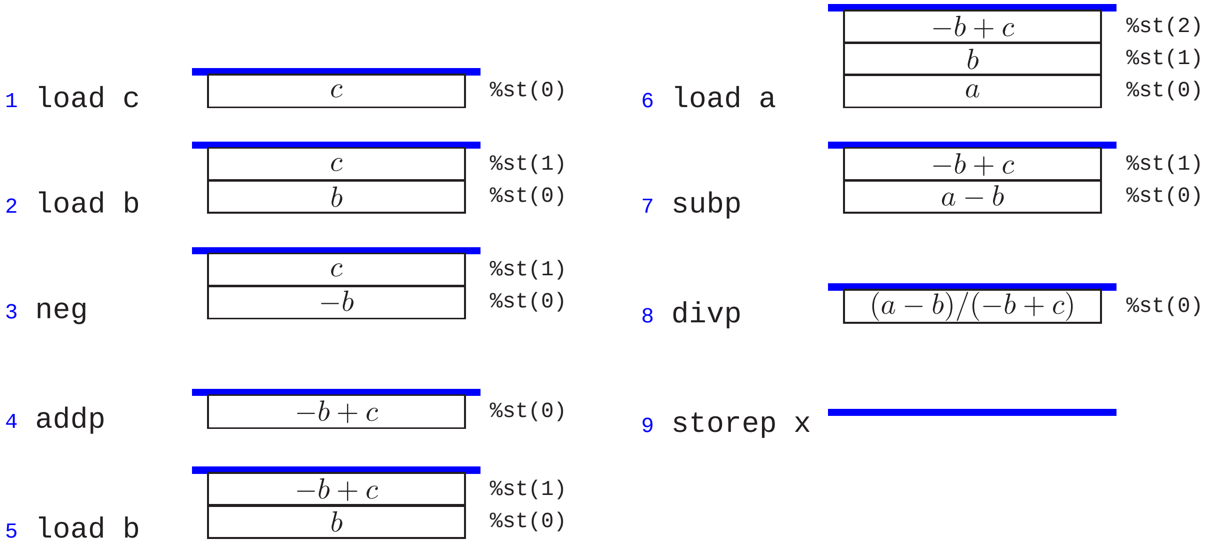
On the other hand, it might be tempting to treat the floating-point registers as a true stack, with each procedure call pushing its local values onto it. Unfortunately, this approach would quickly lead to a stack overflow, since there is room for only eight values. Instead, the x87 registers must be treated as being caller-saved. Compilers generate code that saves every local floating-point value on the main program stack before calling another procedure and then retrieves them on return. This generates memory traffic that can degrade program performance.

As noted in Web Aside DATA:IA32-FP the IA32 floating-point registers are all 80 bits wide. They encode numbers in an *extended-precision* format as described in CS:APP2e Problem 2.85. All single and double-precision numbers are converted to this format as they are loaded from memory into floating-point registers. The arithmetic is always performed in extended precision. Numbers are converted from extended precision to single or double-precision format as they are stored in memory.

3 Stack Evaluation of Expressions

To understand how x87 uses its registers as a stack, let us consider a more abstract version of stack-based evaluation on a hypothetical stack machine. Once we have introduced the basic execution model, we will return to the somewhat more arcane x87 architecture. Assume we have an arithmetic unit that uses a stack to hold intermediate results, having the instruction set illustrated in Figure 1. In addition to the stack, this unit has a memory that can hold values we will refer to by names such as `a`, `b`, and `x`. As Figure 1 indicates, we can push memory values onto this stack with the `load` instruction. The `storep` operation pops the top element from the stack and stores the result in memory. A unary operation such as `neg` (negation) uses the top stack element as its argument and overwrites this element with the result. Binary operations such as `addp` and `multp` use the top two elements of the stack as their arguments. They pop both arguments off the stack and then push the result back onto the stack. We use the suffix '`p`' with the store, add, subtract, multiply, and divide instructions to emphasize the fact that these instructions pop their operands.

As an example, consider the expression `x = (a-b)/(-b+c)`. We could translate this expression into the code that follows. Alongside each line of code, we show the contents of the floating-point register stack. In keeping with our earlier convention, we show the stack as growing downward, so the “top” of the stack is really at the bottom.



```
load c
load b
neg
addp
load b
load a
subp
divp
storep x
```

As this example shows, there is a natural recursive procedure for converting an arithmetic expression into stack code. Our expression notation has four types of expressions having the following translation rules:

1. A variable reference of the form *Var*. This is implemented with the instruction `load` *Var*.
2. A unary operation of the form `-` *Expr*. This is implemented by first generating the code for *Expr* followed by a `neg` instruction.
3. A binary operation of the form $Expr_1$ `+` $Expr_2$, $Expr_1$ `-` $Expr_2$, $Expr_1$ `*` $Expr_2$, or $Expr_1$ `/` $Expr_2$. This is implemented by generating the code for $Expr_2$, followed by the code for $Expr_1$, followed by an `addp`, `subp`, `multp`, or `divp` instruction.
4. An assignment of the form *Var* `=` *Expr*. This is implemented by first generating the code for *Expr*, followed by the `storep` *Var* instruction.

As an example, consider the expression `x = a-b/c`. Since division has precedence over subtraction, this expression can be parenthesized as `x = a-(b/c)`. The recursive procedure would therefore proceed as follows:

1. Generate code for $Expr \doteq$ `a-(b/c)`:
   (a) Generate code for $Expr_2 \doteq$ `b/c`:
      i. Generate code for $Expr_2 \doteq$ `c` using the instruction `load c`.
      ii. Generate code for $Expr_1 \doteq$ `b`, using the instruction `load b`.

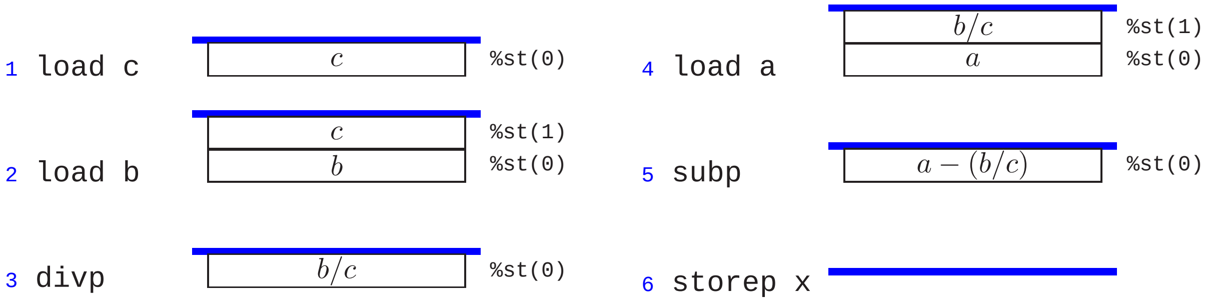
        iii. Generate instruction divp.

    (b) Generate code for $\mathit{Expr}_1 \doteq$ a, using the instruction load a.

    (c) Generate instruction subp.

2. Generate instruction storep x.

The overall effect is to generate the following stack code:

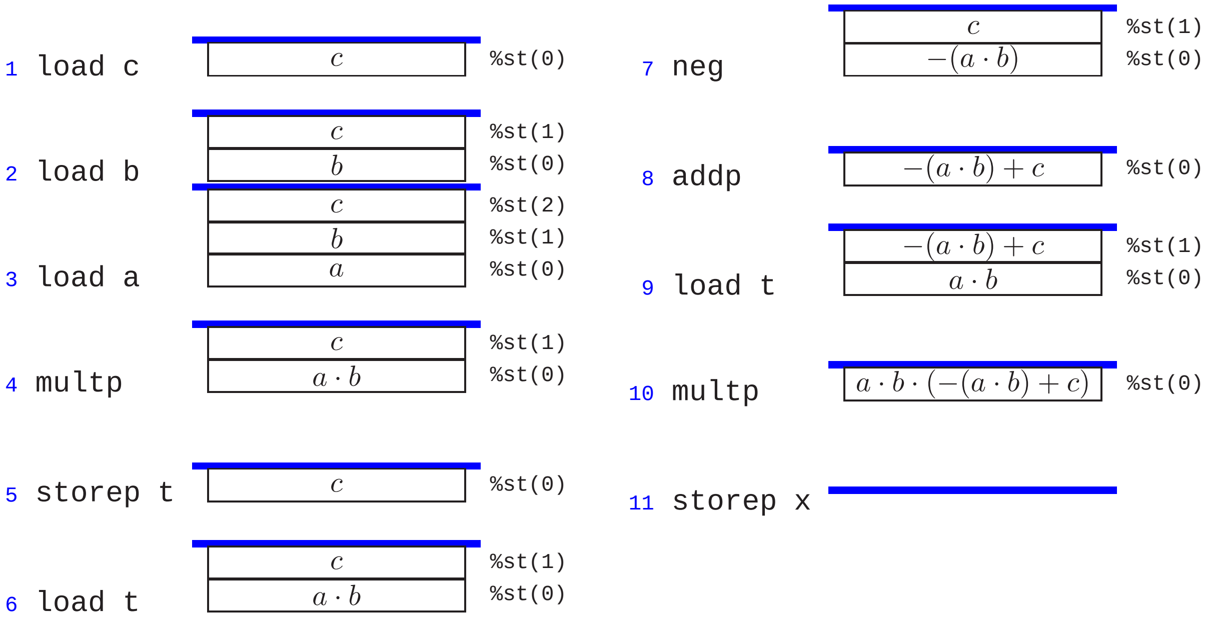


```
load c
load b
divp
load a
subp
storep x
```

Practice Problem 1:

Generate stack code for the expression x = a*b/c * -(a+b*c). Diagram the contents of the stack for each step of your code. Remember to follow the C rules for precedence and associativity.

Stack evaluation becomes more complex when we wish to use the result of some computation multiple times. For example, consider the expression x = (a*b)*(-(a*b)+c). For efficiency, we would like to compute a*b only once, but our stack instructions do not provide a way to keep a value on the stack once it has been used. With the set of instructions listed in Figure 1, we would therefore need to store the intermediate result a*b in some memory location, say t, and retrieve this value for each use. This gives the following code:



```
load c
load b
load a
multp
storep t
load t
neg
addp
load t
multp
storep x
```

Instruction	Source format	Source location
`flds` $Addr$	Single	$\mathsf{M}_4[Addr]$
`fldl` $Addr$	Double	$\mathsf{M}_8[Addr]$
`fldt` $Addr$	Extended	$\mathsf{M}_{10}[Addr]$
`fildl` $Addr$	Integer	$\mathsf{M}_4[Addr]$
`fld` `%st(`i`)`	Extended	`%st(`i`)`

Figure 2: **Floating-point load instructions.** All convert the operand to extended-precision format and push it onto the register stack.

This approach has the disadvantage of generating additional memory traffic, even though the register stack has sufficient capacity to hold its intermediate results. The x87 instruction set avoids this inefficiency by introducing variants of the arithmetic instructions that leave their second operand on the stack, and that can use an arbitrary stack value as their second operand. In addition, it provides an instruction that can swap the top stack element with any other element. Although these extensions can be used to generate more efficient code, the simple and elegant algorithm for translating arithmetic expressions into stack code is lost.

4 Floating-Point Data Movement and Conversion Operations

Floating-point registers are referenced with the notation `%st(`i`)`, where i denotes the position relative to the top of the stack. The value i can range between 0 and 7. Register `%st(0)` is the top stack element, `%st(1)` is the second element, and so on. The top stack element can also be referenced as `%st`. When a new value is pushed onto the stack, the value in register `%st(7)` is lost. When the stack is popped, the new value in `%st(7)` is not predictable. Compilers must generate code that works within the limited capacity of the register stack.

Figure 2 shows the set of instructions used to push values onto the floating-point register stack. The first group of these read from a memory location, where the argument $Addr$ is a memory address given in one of the memory operand formats listed in CS:APP2e Figure 3.3. These instructions differ by the presumed format of the source operand and hence the number of bytes that must be read from memory. Recall that the notation $\mathsf{M}_b[Addr]$ indicates an access of b bytes with starting address $Addr$. All of these instructions convert the operand to extended-precision format before pushing it onto the stack. The final load instruction `fld` is used to duplicate a stack value. That is, it pushes a copy of floating-point register `%st(`i`)` onto the stack. For example, the instruction `fld %st(0)` pushes a copy of the top stack element onto the stack.

Figure 3 shows the instructions that store the top stack element either in memory or in another floating-point register. There are both "popping" versions that pop the top element off the stack (similar to the `storep` instruction for our hypothetical stack evaluator), as well as nonpopping versions that leave the source value on the top of the stack. As with the floating-point load instructions, different variants of the instruction generate different formats for the result and therefore store different numbers of bytes. The first group of these store the result in memory. The address is specified using any of the memory operand formats listed in CS:APP2e Figure 3.3. The second group copies the top stack element to some other floating-point register.

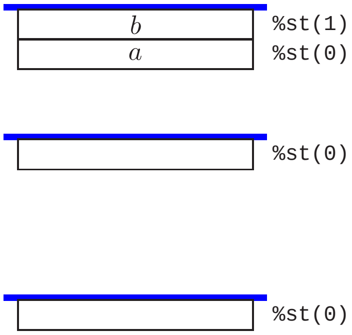
Instruction		Pop (Y/N)	Destination format	Destination location
`fsts`	*Addr*	N	Single	$M_4[Addr]$
`fstps`	*Addr*	Y	Single	$M_4[Addr]$
`fstl`	*Addr*	N	Double	$M_8[Addr]$
`fstpl`	*Addr*	Y	Double	$M_8[Addr]$
`fstt`	*Addr*	N	Extended	$M_{10}[Addr]$
`fstpt`	*Addr*	Y	Extended	$M_{10}[Addr]$
`fistl`	*Addr*	N	Integer	$M_4[Addr]$
`fistpl`	*Addr*	Y	Integer	$M_4[Addr]$
`fst`	`%st(`i`)`	N	Extended	`%st(`i`)`
`fstp`	`%st(`i`)`	Y	Extended	`%st(`i`)`

Figure 3: **Floating-point store instructions.** All convert from extended-precision format to the destination format. Instructions with suffix ‘p’ pop the top element off the stack.

Practice Problem 2:

Assume for the following code fragment that register `%eax` contains an integer variable `x` and that the top two stack elements correspond to variables `a` and `b`, respectively. Fill in the boxes to diagram the stack contents after each instruction

```
    testl %eax,%eax
    jne L11
    fstp %st(0)
    jmp L9
L11:
    fstp %st(1)
L9:
```



Write a C expression describing the contents of the top stack element at the end of this code sequence in terms of `x`, `a` and `b`.

A final floating-point data movement operation allows the contents of two floating-point registers to be swapped. The instruction `fxch %st(`i`)` exchanges the contents of floating-point registers `%st(0)` and `%st(`i`)`. The notation `fxch` written with no argument is equivalent to `fxch %st(1)`, that is, swap the top two stack elements.

Instruction	Computation
fldz	0
fld1	1
fabs	$\|Op\|$
fchs	$-Op$
fcos	$\cos Op$
fsin	$\sin Op$
fsqrt	$\sqrt{Op}$
fadd	$Op_1 + Op_2$
fsub	$Op_1 - Op_2$
fsubr	$Op_2 - Op_1$
fdiv	Op_1 / Op_2
fdivr	Op_2 / Op_1
fmul	$Op_1 \cdot Op_2$

Figure 4: **Floating-point arithmetic operations.** Each of the binary operations has many variants.

Instruction		Operand 1	Operand 2	Format	Destination	Pop %st(0) (Y/N)
fsubs	$Addr$	%st(0)	$\mathrm{M}_4[Addr]$	Single	%st(0)	N
fsubl	$Addr$	%st(0)	$\mathrm{M}_8[Addr]$	Double	%st(0)	N
fsubt	$Addr$	%st(0)	$\mathrm{M}_{10}[Addr]$	Extended	%st(0)	N
fisubl	$Addr$	%st(0)	$\mathrm{M}_4[Addr]$	Integer	%st(0)	N
fsub	%st(i),%st	%st(i)	%st(0)	Extended	%st(0)	N
fsub	%st,%st(i)	%st(0)	%st(i)	Extended	%st(i)	N
fsubp	%st,%st(i)	%st(0)	%st(i)	Extended	%st(i)	Y
fsubp		%st(0)	%st(1)	Extended	%st(1)	Y

Figure 5: **Floating-point subtraction instructions.** All store their results into a floating-point register in extended-precision format. Instructions with suffix ‘p’ pop the top element off the stack.

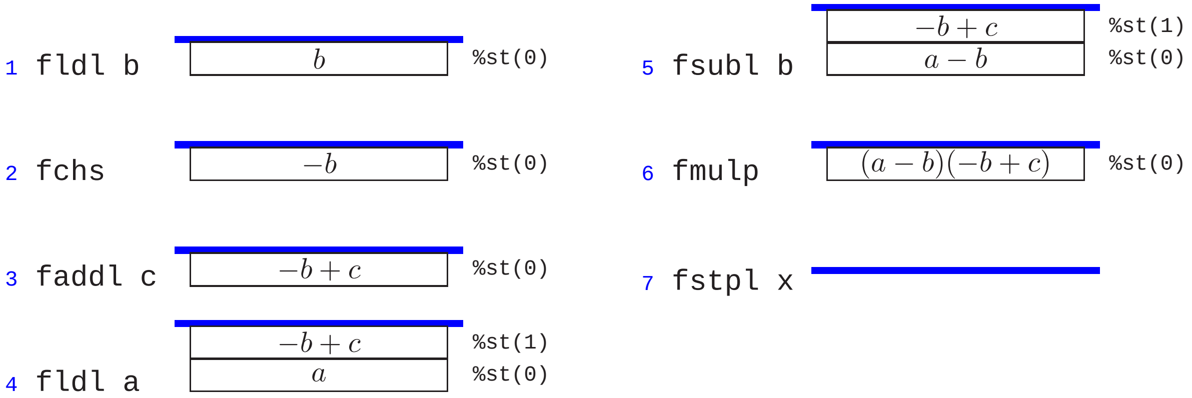
5 Floating-Point Arithmetic Instructions

Figure 4 documents some of the most common floating-point arithmetic operations. Instructions in the first group have no operands. They push the floating-point representation of some numerical constant onto the stack. There are similar instructions for such constants as π, e, and $\log_2 10$. Instructions in the second group have a single operand. The operand is always the top stack element, similar to the `neg` operation of the hypothetical stack evaluator. They replace this element with the computed result. Instructions in the third group have two operands. For each of these instructions, there are many different variants for how the operands are specified, as will be discussed shortly. For noncommutative operations such as subtraction and division there is both a forward (e.g., `fsub`) and a reverse (e.g., `fsubr`) version, so that the arguments can be used in either order.

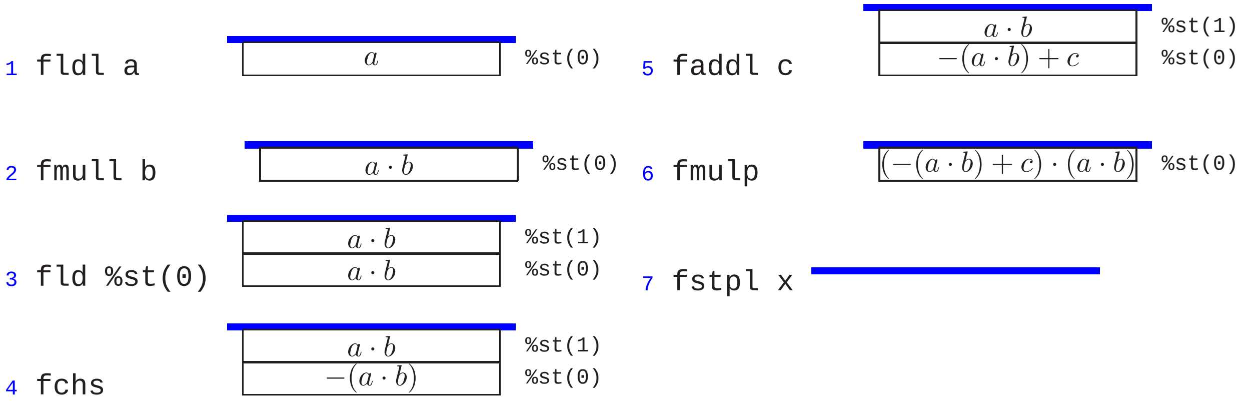
In Figure 4 we show just a single form of the subtraction operation `fsub`. In fact, this operation comes in many different variants, as shown in Figure 5. All compute the difference of two operands: $Op_1 - Op_2$ and store the result in some floating-point register. Beyond the simple `subp` instruction we considered for the hypothetical stack evaluator, x87 has instructions that read their second operand from memory or from some floating-point register other than `%st(1)`. In addition, there are both popping and nonpopping variants. The first group of instructions reads the second operand from memory, either in single-precision, double-precision, or integer format. It then converts this to extended-precision format, subtracts it from the top stack element, and overwrites the top stack element. These can be seen as a combination of a floating-point load following by a stack-based subtraction operation.

The second group of subtraction instructions use the top stack element as one argument and some other stack element as the other, but they vary in the argument ordering, the result destination, and whether or not they pop the top stack element. Observe that the assembly code line `fsubp` is shorthand for `fsubp %st,%st(1)`. This line corresponds to the `subp` instruction of our hypothetical stack evaluator. That is, it computes the difference between the top two stack elements, storing the result in `%st(1)`, and then popping `%st(0)` so that the computed value ends up on the top of the stack.

All of the binary operations listed in Figure 4 come in all of the variants listed for `fsub` in Figure 5. As an example, we can write the code for the expression `x = (a-b)*(-b+c)` using x87 instructions. For exposition purposes we will still use symbolic names for memory locations and we assume these are double-precision values.



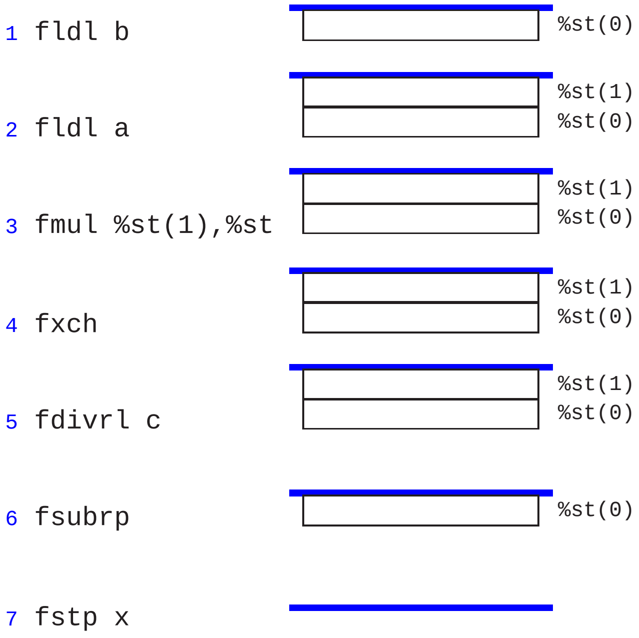
As another example, consider the expression `x = ((-(a*b)+c)*(a*b)`. Observe how the instruction `fld %st(0)` is used to create two copies of `a*b` on the stack, avoiding the need to save the value in a temporary memory location.



```
fldl a
fmull b
fld %st(0)
fchs
faddl c
fmulp
fstpl x
```

Practice Problem 3:

Diagram the stack contents after each step of the following code:



```
fldl b
fldl a
fmul %st(1),%st
fxch
fdivrl c
fsubrp
fstp x
```

Give a C expression describing this computation.

6 Using Floating Point in Procedures

With IA32, floating-point arguments are passed to a calling procedure on the stack, just as are integer arguments. Each parameter of type `float` requires 4 bytes of stack space, while each parameter of type double requires 8. For functions whose return values are of type float or double, the result is returned on the top of the floating-point register stack in extended-precision format.

As an example, consider the following function

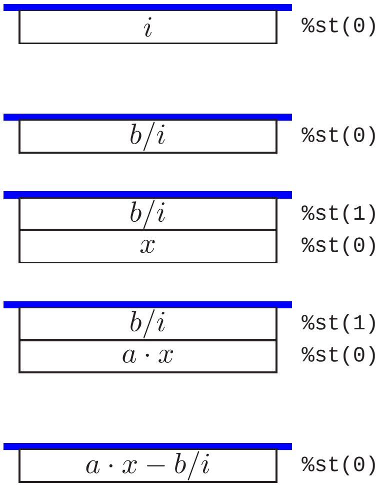
```
double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}
```

Arguments a, x, b, and i will be at byte offsets 8, 16, 20, and 28 relative to %ebp, respectively, as follows:

Offset	8	16	20	28
Contents	a	x	b	i

The body of the generated code, and the resulting stack values are as follows:

```
fildl 28(%ebp)
fdivrl 20(%ebp)
flds 16(%ebp)
fmull 8(%ebp)
fsubp %st,%st(1)
```



Practice Problem 4:

For a function funct2 with arguments p, q, r, and s, the compiler generates the following code for the function body:

```
fildl   8(%ebp)
flds    20(%ebp)
faddl   12(%ebp)
fdivrp  %st, %st(1)
fld1
fadds   24(%ebp)
fsubrp  %st, %st(1)
```

The returned value is of type double. Write C code for funct2. Be sure to correctly declare the argument types.

Ordered	Unordered	Op_2	Type	Number of pops
fcoms $Addr$	fucoms $Addr$	$\mathsf{M}_4[Addr]$	Single	0
fcoml $Addr$	fucoml $Addr$	$\mathsf{M}_8[Addr]$	Double	0
fcom %st(i)	fucom %st(i)	%st(i)	Extended	0
fcom	fucom	%st(1)	Extended	0
fcomps $Addr$	fucomps $Addr$	$\mathsf{M}_4[Addr]$	Single	1
fcompl $Addr$	fucompl $Addr$	$\mathsf{M}_8[Addr]$	Double	1
fcomp %st(i)	fucomp %st(i)	%st(i)	Extended	1
fcomp	fucomp	%st(1)	Extended	1
fcompp	fucompp	%st(1)	Extended	2

Figure 6: **Floating-point comparison instructions.** Ordered vs. unordered comparisons differ in their treatment of *NaN*'s.

$Op_1 : Op_2$	Binary	Decimal
$>$	[00000000]	0
$<$	[00000001]	1
$=$	[01000000]	64
Unordered	[01000101]	69

Figure 7: **Encoded results from floating-point comparison.** The results are encoded in the high-order byte of the floating-point status word after masking out all but bits 0, 2, and 6.

7 Testing and Comparing Floating-Point Values

Similar to the integer case, determining the relative values of two floating-point numbers involves using a comparison instruction to set condition codes and then testing these condition codes. For floating point, however, the condition codes are part of the *floating-point status word*, a 16-bit register that contains several flags about the floating-point unit. This status word must be transferred to an integer word, and then the particular bits must be tested.

There are a number of different floating-point comparison instructions as documented in Figure 6. All of them perform a comparison between operands Op_1 and Op_2, where Op_1 is the top stack element. Each line of the table documents two different comparison types: an "ordered" comparison and an "unordered" comparison. The two comparisons differ only in how they handle the case when both arguments are some form of *NaN*. Even then, their only difference are that one sets an exception flag while the other does not, but this flag is typically ignored anyhow, and so we find GCC using the two forms interchangeably.

Different comparison instructions also differ in the location of operand Op_2, analogous to the different forms of floating-point load and floating-point arithmetic instructions. Finally, the different forms differ in the number of elements popped off the stack after the comparison is completed. Instructions in the first group shown in the table do not change the stack at all. Even for the case where one of the arguments is in memory, this value is not on the stack at the end. Operations in the second group pop element Op_1 off the stack. The final operation pops both Op_1 and Op_2 off the stack.

The floating-point status word is transferred to an integer register with the `fnstsw` instruction. The operand for this instruction is one of the 16-bit register identifiers shown in CS:APP2e Figure 3.2, for example, `%ax`. The bits in the status word encoding the comparison results are in bit positions 0, 2, and 6 of the high-order byte of the status word. For example, if we use instruction `fnstw %ax` to transfer the status word, then the relevant bits will be in `%ah`. A typical code sequence to select these bits is then:

```
fnstsw  %ax                    Store floating-point status word in %ax
testb   $69, %ah               Test bits 0, 2, and 6  of word
```

Note that 69_{10} has bit representation $[01000101]$, that is, it has 1s in the three relevant bit positions. Figure 7 shows the possible values of byte `%ah` that would result from this code sequence. Observe that there are only four possible outcomes for comparing operands Op_1 and Op_2: the first is either greater, less, equal, or incomparable to the second, where the latter outcome only occurs when one of the values is a NaN. (Any comparison with a *NaN* value should yield 0. For example, if `x` is *NaN*, then the comparisons `x < y`, `x == y`, and `x > y` should all yield 0.)

As an example, consider the following procedure:

```
int less(double x, double y)
{
    return x < y;
}
```

The compiled code for the function body is as follows:

```
fldl    16(%ebp)              Push y
fldl    8(%ebp)               Push x
fxch    %st(1)                Swap x and y on stack
fucompp                       Compare y:x and pop both
fnstsw  %ax                   Store floating-point status word in %ax
testb   $69, %ah              Test bits 0, 2, and 6  of word
sete    %al                   Test for comparison outcome of 0 (>)
movzbl  %al, %eax             Zero extend to get result
```

Practice Problem 5:

Suppose we want to generate code for the following function

```
int lesseq(double x, double y)
{
    return x <= y;
}
```

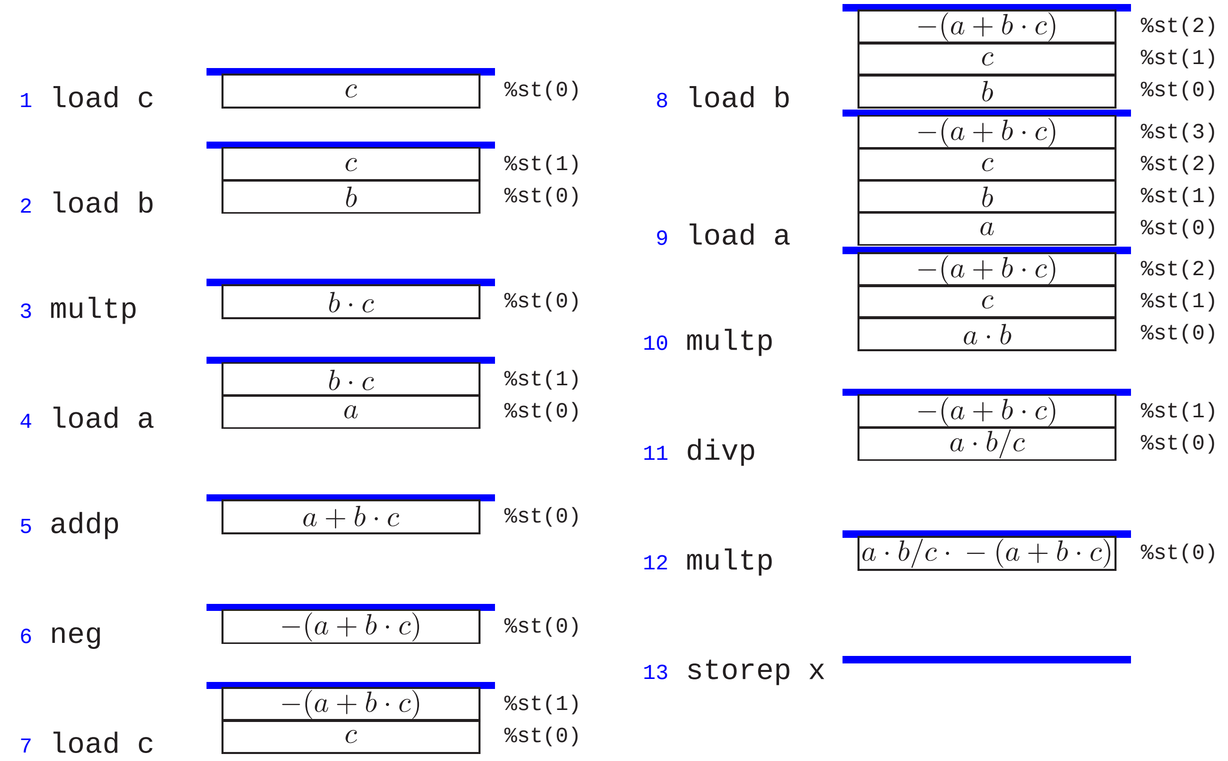
How could we adapt the code generated for the `less` function, changing only the mask used by the `testb` instruction from 69 to something else.

This completes our coverage of floating-point programming with x87. Even experienced programmers find this code arcane and difficult to read. The stack-based operations, the awkwardness of getting status results from the FPU to the main processor, and the many subtleties of floating-point computations combine to make the machine code lengthy and obscure. As mentioned in the introduction, the SSE floating-point architecture has become the preferred method for implementing floating-point operations on x86 processors, but we can expect that the use of x87 instructions will continue, given the large amount of legacy code and the many legacy machines that still exist.

Solutions to Problems

Problem 1 Solution: [Pg. 5]

This problem gives you a chance to try out the recursive procedure described in Section 3.

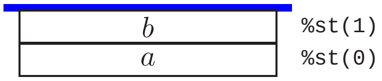


```
load c
load b
multp
load a
addp
neg
load c
load b
load a
multp
divp
multp
storep x
```

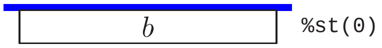
Problem 2 Solution: [Pg. 6]

The following code is similar to that generated by the compiler for selecting between two values based on the outcome of a test:

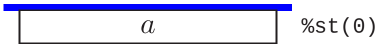
```
      test %eax,%eax
      jne L11
```



```
    fstp %st(0)
```



```
    jmp L9
L11:
    fstp %st(1)
```



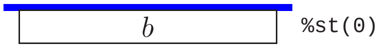
```
L9:
```

The resulting top of stack value is `x ? a : b`.

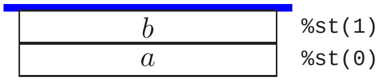
Problem 3 Solution: [Pg. 10]

Floating-point code is tricky, with its different conventions about popping operands, the order of the arguments, etc. This problem gives you a chance to work through some specific cases in complete detail.

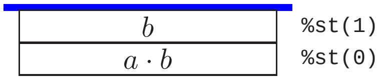
```
fldl b
```



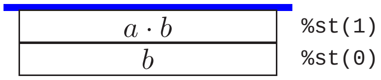
```
fldl a
```



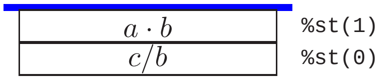
```
fmul %st(1),%st
```



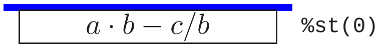
```
fxch
```



```
fdivrl c
```



```
fsubrp
```



```
fstp x
```

This code computes the expression `x = a*b - c/b`.

Problem 4 Solution: [Pg. 11]

This problem requires you to think about the different operand types and sizes in floating-point code. Looking at the code, we see that it reads its arguments from offsets 8, 12, 20, and 24 relative to `%ebp`.

We then annotate the code as follows:

```
  p, q, r, and s are at offsets 8, 12, 16, and 24 from %ebp
fildl   8(%ebp)          Get p (int)
flds    20(%ebp)         Get r (float)
faddl   12(%ebp)         Get q (double) and compute q+r
fdivrp  %st, %st(1)      Compute p/(q+r)
fld1                     Load 1.0
fadds   24(%ebp)         Get s (float) and compute s+1.0
fsubrp  %st, %st(1)      Compute p/(q+r) - (s+1.0)
```

Based on the instructions that read the arguments from memory, we determine that arguments `p`, `q`, `r`, and `s` are of types `int`, `double`, `float`, and `float`, respectively. We can also determine the expression being computed, leading us to generate the following C version of the code:

```
double funct2(int p, double q, float r, float s)
{
    return p/(q+r) - (s+1);
}
```

Problem 5 Solution: [Pg. 13]

Since the arguments are compared in reverse order, we want to return 1 when the outcome of the test is either greater or equal. The code should accept the two cases in Figure 7 with decimal values 0 and 64 but reject those with values 1 and 69. Using a mask of either 1 or 5 will accomplish this.

Index