Some notes on the new MLRISC X86 floating point code generator (Draft)

Allen Leung[*] Lal George[†]

1 Introduction

Generating good code for the Intel x86 architecture is a difficult task in general, and is especially difficult for floating point intensive programs. The main reason for this is that floating point instructions on the x86 are not register oriented, which preclude many kinds of traditional analysis and optimizations. In particular, the x86 uses a hardware floating point stack for floating point arithmetic, and all floating point instructions are stack based.

The obvious and easiest way to generate floating code on the x86 is to take advantage of the stack orientation:

1. Map virtual registers onto memory locations,
2. Translate floating point expressions into the appropriate stack based instructions, via Sethi-Ullman numbering [SU70], or something more advanced that take into account of latencies [PF91].

While this uses the floating point ISA in the natural way, it does not produce the fastest code. First, the top of stack can become the bottleneck, limiting the amount of pipelining and out-of-order execution possible. Secondly, extra memory transfers may have to performed if all variables are mapped into memory. Again, the presence of memory transfer operations further limit pipelining and out-of-order execution. Furthermore, this style of code generation tends to move temporaries in and out of the floating point stack unnecessarily. Lastly, large expressions that can take full advantage of the floating point stack (8 levels deep) are seldomly found in programs, and the floating point stack is frequently not fully utilized.

Ironically, the preferred code generation methodology advocated by the Intel optimization guide [Cor97] is to treat the floating point stack as a bank of 8 registers, and use the `fxch` instruction to swap contents of the stack as needed. The instruction `fxch` is implemented with register renaming internally by the processor and incurs no cost if paired with most floating point instructions.

[*]New York University. Email: `leunga@cs.nyu.edu`
[†]Bell Labs. Email: `george@research.bell-labs.com`

Thus to fully take advantage of the x86 architecture, it is necessary to "bypass" the stack-based ISA and simulate the internal micro-architecture. This is achieved in the optimized mode of the latest MLRISC floating point code generator. Under this mode, the following code generation scheme is used:

1. First, we assume that there are 7 physical floating point registers, `%f0` ... `%f6`, available to the program, with an additional hidden register available for scratch.

2. The code generate emits pseudo 3-address instructions that do not mention the floating point stack.

3. During register allocation[1], virtual registers are mapped onto `%f0` ... `%f6`. Because of the dearth of registers, memory coalescing is also enabled to optimize spill code.

4. Finally, we invoke a rewriting phase to translate the 3-address pseudo instructions into actual stack based instructions. During this phase the mapping from the pseudo registers `%f0` ... `%f6` and the stack locations `%st(0)` ... `%st(7)` are recomputed at each program point. The register allocation phase guarantees that this recomputation is always possible because at most 7 values are live at any program point. As expected, most of the implementation complexity is in this last phase.

1.1 The X86 floating point instruction set

Figure 1 lists some common x86 floating point instructions. All floating point instructions operate directly on the floating point stack, using the top of the stack as the accumulator. The stack locations are numbered from `%st(0)` to `%st(7)`, where `%st(0)` is the top of the stack.

1.2 The pseudo instruction set

Figure 2 lists the corresponding pseudo 3-address floating point instructions. All pseudo instructions can take virtual registers or memory effective addresses as sources or destinations. One exception is the instruction `fcmp`, which must have at least one register argument. This restriction is necessary because of irregularity in the instructions `fucom`, `fucomp`, `fucompp`, which cannot take a memory argument.

[1]The *bandwidth* of a program is the maximal number of live-ranges that are simultaneously live at all program points. Actually, what we require is some method to reduce the bandwidth of the program to 7. It is not exactly the same as register allocation, but that is a close enough approximation.

Instruction (gas syntax)	Semantics
`fsub %st(n), %st`	`%st(0)` := `%st(0)` − `%st(n)`
`fsub %st, %st(n)`	`%st(n)` := `%st(0)` − `%st(n)`
`fsubr %st(n), %st`	`%st(0)` := `%st(n)` − `%st(0)`
`fsubr %st, %st(n)`	`%st(n)` := `%st(n)` − `%st(0)`
`fsubp %st, %st(n)`	`%st(n)` := `%st(0)` − `%st(n)`; pop stack
`fsubrp %st, %st(n)`	`%st(n)` := `%st(n)` − `%st(0)`; pop stack
`fsub mem`	`%st(0)` := `%st(0)` − *mem*
`fsubr mem`	`%st(0)` := *mem* − `%st(0)`
`fsqrt`	`%st(0)` := *sqrt*(`%st(0)`)
`fldl mem`	load mem, and push it onto `%st(0)`
`fldl %st(n)`	push `%st(n)` onto `%st(0)`
`fstl mem`	store `%st(0)` into mem
`fstpl mem`	store `%st(0)` into mem and pop stack
`fst %st(n)`	`%st(n)` := `%st(0)`
`fstp %st(n)`	`%st(n)` := `%st(0)`; pop stack
`fxch %st(n)`	exchange `%st(0)` and `%st(n)`
`fucom %st(n)`	compare `%st(0)` and `%st(n)`; set status
`fucomp %st(n)`	compare `%st(0)` and `%st(n)`; set status; pop
`fucompp`	compare `%st(0)` and `%st(1)`; set status; pop twice

Figure 1: Some X86 floating point instructions and their semantics

Instruction	Semantics
`fsubl src1, src2, dst`	dst := src1 - src2
`fsubrl src1, src2, dst`	dst := src2 - src1
`fisubl src1, src2, dst`	dst := src1 - real(src2)
`fisubrl src1, src2, dst`	dst := real(src2) - src1
`fmovel src, dst`	dst := src
`fiload src, dst`	dst := real(src)
`fcmp src1, src2`	compare src1 and src2 and set status

Figure 2: Pseudo floating point instructions and their semantics

1.3 A Basic Block Example

For example, suppose we want to generate code for the following floating point statements:

$$\begin{aligned} a &:= x * x \\ b &:= y * y \\ c &:= z * z \\ d &:= sqrt(a + b + c) \\ w &:= d/(a - b - c) \end{aligned}$$

The naive but direct way of implementing these statements in the x86 architecture is to map the variables `a, b, c, d, x, y, z` and `w` into some memory locations[2]. The following sequence of code is generated:

Instruction	Stack
fldl x	[x]
fmull x	[x*x]
fstpl a	[]
fldl y	[y]
fmull y	[y*y]
fstpl b	[]
fldl z	[z]
fmull z	[z*z]
fstpl c	[]
fldl a	[a]
faddl b	[a+b]
faddl c	[a+b+c]
fsqrt	[sqrt(a+b+c)]
fldl a	[a]
fsubl b	[a-b]
fsubl c	[a-b-c]
fdivr d	[d/(a-b-c)]
fstpl w	[]

However, note that the three multiply instructions cannot be pipelined because of the dependency on `%st(0)`. In fact, no floating point instructions can be pipelined at all in this sequence! Furthermore, all instructions (except `fsqrt`) require an explicit reference to memory, which in the best case, require an access to the first level cache.

In the more advanced scheme we will first generate the following sequence of pseudo instructions:

```
fmove x, %f0
fmove y, %f1
fmove z, %f2
```

[2]Register allocation can be applied to minimize the number of memory locations needed. But we will ignore this detail for now.

```
fmull %f0, %f0, $f4
fmull %f1, %f1, %f5
fmull %f2, %f2, %f6
faddl %f4, %f5, %f7
faddl %f7, %f6, %f8
fsqrt %f8, %f9
fsubl %f4, %f5, %f10
fsubl %f10, %f6, %f11
fdivl %f9, %f11, %f12
fmove %f12, w
```

Note that we use the common MLRISC code generation idiom: we generate parallel copies so that all values in memory locations, i.e., `x, y` and `z`, are first transfered to temporaries. We depend on the register allocator to optimize out these register promotions when appropriate [LG00].

After register allocation, we obtain this sequence of code:

```
fmove x, %f0
fmove y, %f1
fmove z, %f2
fmull %f0, %f0, $f0
fmull %f1, %f1, %f1
fmull %f2, %f2, %f2
faddl %f0, %f1, %f3
faddl %f3, %f2, %f3
fsqrt %f3, %f3
fsubl %f0, %f1, %f4
fsubl %f4, %f2, %f4
fdivl %f3, %f4, %f5
fmove %f5, w
```

A rewriting phase will then translate the above into the following stack based instructions.

Instruction	Stack
`fldl x`	[x]
`fldl y`	[y x]
`fldl z`	[z y x]
`fxch %st(2)`	[x y z]
`fmul %st(0), %st`	[x*x y z]
`fxch %st(2)`	[y x*x z]
`fmul %st(0), %st`	[y*y x*x z]
`fxch %st(2)`	[z x*x y*y]
`fmul %st(0), %st`	[z*z y*y x*x]
`fld %st(2)`	[x*x z*z y*y x*x]
`fadd %st(2), %st`	[x*x+y*y z*z y*y x*x]
`fadd %st(1), %st`	[x*x+y*y+z*z z*z y*y x*x]
`fsqrt`	[sqrt(x*x+y*y+z*z) z*z y*y x*x]
`fxch %st(3)`	[x*x z*z y*y sqrt(x*x+y*y+z*z)]
`fsubrp %st, %st(2)`	[z*z x*x-y*y sqrt(x*x+y*y+z*z)]
`fsubrp %st, %st(1)`	[x*x-y*y-z*z sqrt(x*x+y*y+z*z)]
`fdivr %st, %st(1)`	[sqrt(x*x+y*y+z*z)/(x*x-y*y-z*z)]
`fstpl w`	[]

In this sequence, the three multiply instructions can be executed in a pipelined fashion, since the inserted `fxch` instructions have removed the dependency on `%st(0)`. During execution, the `fxch` instructions are paired with the `fmul` and incur no pipeline delays. Aside from the initially loads and the final stores (which we assume to be unavoidable), all references to memory have also been removed. Another side advantage of this sequence is that it has a more compact encoding than the previous one, since instructions referencing the floating point stack have 2-byte encodings, while instructions that reference memory operands require at least 3 bytes to encode.

2 Floating Point Register Allocation

Floating point register allocation of pseudo instructions proceeds in essentially the same manner as the integer component, as described in [LG00, LG99]. We use the iterated coalescing algorithm of George and Appel [GA96] for coalescing copies in registers. Spill coalescing, propagation and coloring phases, as described in [Geo99], are used to minimize the amount of memory references.

3 Rewriting Pseudo Instructions

After register allocation, we rewrite the pseudo instructions into the correct stack oriented instructions. This is done in the following steps.

1. At each program point, we first compute the set of registers that are used last and those that are dead[3]. This requires a pass of liveness analysis.

[3] The latter is necessary if there are dead code.

However, if we run the rewriting phase immediately after register allocation, this information is available for free.

2. Next, we rewrite the instructions in each basic block one at one. During this rewriting process, we compute the bindings between the pseudo `%f`m registers and the `%st(`n`)` stack registers. It should be emphasized that this computation is done by simulating the x86 stack program point by program point, and we do not try to use a fixed, global mapping between `%f`n registers and `%st(`n`)` stack registers.

3. Because the incoming bindings and outgoing bindings of two connecting blocks may be different, we may have to insert compensation code at a flowgraph edge that connects two blocks. Such code performs two functions:

   (a) Deallocate all dead stack entries, and

   (b) Shuffle the remaining bindings to the correct positions of the stack.

   Insertion of compensation code at edges can be simulated by insertion to nodes, as is commonly known (e.g. [DRZ92]); however, in the presence of *critical edges*, i.e., edges connecting a split node to a merge node, splitting may be required.

3.1 How to Rewrite Instructions

The rewriting rules that we use try to reclaim dead `%st(`n`)` registers as soon as possible. We use a local and greedy approach which minimizes the number of `fxch` and `fld` instructions that are inserted at each step. Note that this approach is not optimal with respect to the total number of extra instructions inserted, because it does not take into account of global information. However, it can be implemented very efficiently and simply.

Notationally, a rewriting rule is written in the following form:

$$\langle S, \mathtt{x}\rangle \longrightarrow \langle S', \mathtt{y}\rangle$$

with possibly side conditions. This means that the pseudo instruction `x`, given the current stack bindings S, should be translated into the real instruction(s) `y`, resulting in a new stack S'.

Operations on a stack and their semantics are listed below.

Operation	Semantics
$\mathsf{pop}(S)$	pop stack S
$\mathsf{push}(S, \%\mathtt{fr})$	push `%fr` onto the top of the stack
$S\{n \to \%\mathtt{fr}\}$	stack location `%st(n)` is bound to register `%fr`
$\mathsf{xch}(S, n)$	Exchange stack location `%st(0)` and `%st(n)`

Side conditions are of the following three forms:

Side condition	Meaning
$S \vdash \texttt{\%fx} \mapsto \texttt{\%st(n)}$	%fx is currently located in %st(n) in S
last %fx	%fx is the last use at this program pt.
live %fx	%fx is non-the last use at this program pt.

Some translation rules for binary operators are listed in Figure 3.

$$\textbf{(1)}\quad \frac{\begin{array}{c} S \vdash \texttt{\%f}x \mapsto \texttt{\%st}(0), \texttt{\%f}y \mapsto \texttt{\%st}(n) \\ S \vdash \mathsf{last}\ \texttt{\%f}x, \mathsf{live}\ \texttt{\%f}y \end{array}}{\langle S, \texttt{fsub \%f}x\texttt{, \%f}y\texttt{, \%f}z\rangle \longrightarrow \langle S\{0 \to \texttt{\%f}z\}, \texttt{fsub \%st}(n)\texttt{, \%st}\rangle}$$

$$\textbf{(2)}\quad \frac{\begin{array}{c} S \vdash \texttt{\%f}x \mapsto \texttt{\%st}(m), \texttt{\%f}y \mapsto \texttt{\%st}(n) \\ S \vdash \mathsf{live}\ \texttt{\%f}x, \mathsf{live}\ \texttt{\%f}y \end{array}}{\begin{array}{c} \langle S, \texttt{fsub \%f}x\texttt{, \%f}y\texttt{, \%f}z\rangle \\ \longrightarrow \\ \langle \mathsf{push}(S, \texttt{\%f}z), \texttt{fld \%st}(m)\texttt{; fsub \%st}(n+1)\texttt{, \%st}\rangle \end{array}}$$

$$\textbf{(3)}\quad \frac{\begin{array}{c} S \vdash \texttt{\%f}x \mapsto \texttt{\%st}(0), \texttt{\%f}y \mapsto \texttt{\%st}(n) \\ S \vdash \mathsf{last}\ \texttt{\%f}x, \mathsf{last}\ \texttt{\%f}y \end{array}}{\langle S, \texttt{fsub \%f}x\texttt{, \%f}y\texttt{, \%f}z\rangle \longrightarrow \langle \mathsf{pop}(S\{n \to \texttt{\%f}z\}), \texttt{fsubrp \%st, \%st}(n)\rangle}$$

$$\textbf{(4)}\quad \frac{\begin{array}{c} S \vdash \texttt{\%f}x \mapsto \texttt{\%st}(0), \texttt{\%f}y \mapsto \texttt{\%st}(n) \\ S \vdash \mathsf{live}\ \texttt{\%f}x, \mathsf{last}\ \texttt{\%f}y \end{array}}{\langle S, \texttt{fsub \%f}x\texttt{, \%f}y\texttt{, \%f}z\rangle \longrightarrow \langle S\{n \to \texttt{\%f}z\}, \texttt{fsubr \%st, \%st}(n)\rangle}$$

$$\textbf{(5)}\quad \frac{\begin{array}{c} S \vdash \texttt{\%f}x \mapsto \texttt{\%st}(m), \texttt{\%f}y \mapsto \texttt{\%st}(n), m \neq 0 \\ S \vdash \mathsf{last}\ \texttt{\%f}x, \mathsf{live}\ \texttt{\%f}y \\ \langle \mathsf{xch}(S, m), \texttt{x}\rangle \longrightarrow \langle \mathsf{xch}(S, m), \texttt{y}\rangle \end{array}}{\langle S, \texttt{x}\rangle \longrightarrow \langle \mathsf{xch}(S, m), \texttt{fxch \%st}(m)\texttt{; y}\rangle}$$

Figure 3: Some translation rules for binary operators.

For example, suppose that current stack contains the following set of bindings

$$[\texttt{\%st(0)} \to \texttt{\%f0}, \texttt{\%st(1)} \to \texttt{\%f1}, \texttt{\%st(2)} \to \texttt{\%f2}]$$

Then the instruction

```
fsub %f0,%f1,%f3
```

is translated into the

```
fsub %st(1)
```

and resulting in the new stack

$$[\texttt{\%st(0)} \to \texttt{\%f3}, \texttt{\%st(1)} \to \texttt{\%f1}, \texttt{\%st(2)} \to \texttt{\%f2}]$$

if %f0 is dead and %f1 is not after this instruction. On the other hand, if both %f0 and %f1 are live after this instruction, we translate the pseudo instruction into the following sequence:

```
fld  %st(0)
fsub %st(2)
```

and resulting in the new stack

$$[\texttt{\%st(0)} \to \texttt{\%f3}, \texttt{\%st(1)} \to \texttt{\%f0}, \texttt{\%st(2)} \to \texttt{\%f1}, \texttt{\%st(3)} \to \texttt{\%f2}]$$

Similarly, given the pseudo instruction

$$\texttt{fsub \%f1}, \texttt{\%f2}, \texttt{\%f3}$$

where %f1 is the last use, we will apply rule (5) to swap the contents of %f1 into the top of the stack. The resulting instruction sequence is:

```
fxch %st(1)
fsub %st(2)
```

3.2 Rewriting parallel copies

Parallel copies that remain in the program after register allocation can usually be simulated by renaming without emitting any code. For example, suppose we are given the parallel copy

$$\texttt{\%f1} \to \texttt{\%f2}, \texttt{\%f2} \to \texttt{\%f1}$$

we can simply swap the current bindings for %f1 and %f2[4].

The general code generation algorithm is as follows. When we are given a parallel copy of the form:

$$\begin{array}{ccc} \texttt{\%f}d_1 & \to & \texttt{\%f}s_1 \\ & \cdots & \\ \texttt{\%f}d_k & \to & \texttt{\%f}s_k \end{array}$$

we classify each source register $\texttt{\%f}s_i$ as last use or non-last use. Note that if the same source register appears multiple times as sources then only once instance of which can be a last use. Non-last use copies are implemented by explicitly copying the value. Last-use copies are implemented by renaming.

For example, suppose we are given a parallel copy:

$$\begin{array}{ccc} \texttt{\%f2} & \to & \texttt{\%f0} \\ \texttt{\%f3} & \to & \texttt{\%f1} \\ \texttt{\%f4} & \to & \texttt{\%f1} \end{array}$$

[4] In fact, because renaming is for free, coalescing of floating point registers is not a crucial factor during the register allocation phase.

and the current stack is `[%f0, %f1]`. Further suppose that the values in `%f0` and `%f1` are dead after the parallel copy. Then we generate

```
fld %st(1)
```

for the copy `%f3` $\rightarrow$ `%f1`. The renaming copies can be simulated by renaming the stack entries. In this example, the final stack contains `[%f3, %f2, %f1]`.

3.3 Generating stack deallocation code

Stack deallocation code are inserted at the beginning of each basic block, at compensation edges, and, if dead code can be present in the program, after each instruction[5]. For example, if the stack contains the following values: `[%f1, %f0, %f2, %f4]` and `%f2` and `%f4` are dead, then the following sequence of pops are generated:

```
fstp %st(2)
fstp %st(2)
```

The resulting stack will contain the following values: `[%f1, %f0]`

3.4 Generating shuffle code

Frequently, we need to insert shuffle code to connect two basic blocks because values are at different positions on the stack. For example, at the end of a basic block the floating point stack may contain the following values `[%f0, %f2, %f1]`, while the connecting block may require the values to be stacked differently, say `[%f1, %f0, %f2]`. We emit `fxch` instructions for this purposes.

First, it is clear that this problem of shuffle code generation is identical to decomposing a permutation into exchanges of the form $(0, n)$, i.e., exchanges between the first position and the nth position.

We use the following algorithm for this purpose. Given a permutation P, we first decompose P into its (non-trivial) cycles $C_1, \ldots, C_k$. We distinguish between cycles that involve the element 0, and those that do not involve the element 0.

For a cycle $(a_1, \ldots, a_k)$ of the latter form, we generate the following sequence of exchanges.

```
fxch %st(a_1)
fxch %st(a_2)
...
fxch %st(a_k)
fxch %st(a_1)
```

For a cycle $(a_1, \ldots, a_k)$ that involves the element 0, we first normalize this into an equivalent cycle $(a'_1, \ldots, a'_{k-1}, 0)$. The following sequence of code is then generated.

[5] Alternatively, we can also suppress the generation of dead code.

```
fxch %st(a'_1)
fxch %st(a'_2)
...
fxch %st(a'_{k-1})
```

A linear-time algorithm for generating shuffle code can be implemented in the obvious manner. Furthermore, the number of generated exchanges is optimal (proof omitted.)

For example, suppose the initial bindings of the floating point stack is

```
[%st(0)=%f2 %st(1)=%f3 %st(2)=%f4 %st(3)=%f1 %st(4)=f0]
```

and we would like to rearrange the bindings to match:

```
[%st(0)=%f3 %st(1)=%f2 %st(2)=%f0 %st(3)=%f1 %st(4)=f4]
```

Here is how we proceed. First, we compute the permutation that characterizes this rearrangement, which is

$$[0 \to 1, 1 \to 0, 2 \to 4, 3 \to 3, 4 \to 2]$$

This permutation can be decomposed into the following two cycles

$$(0,1)(2,4)$$

We first generate code to compute the cycle $(2,4)$:

```
fxch %st(4)
fxch %st(2)
fxch %st(4)
```

followed by the code to compute the cycle $(0,1)$:

```
fxch %st(1)
```

3.5 Bindings Propagation

To eliminate unnecessary compensation code between basic blocks, we use the following simple heuristics to propagate stack bindings from one basic block to another. Given a block A, with successors blocks $\mathsf{B}_1, \ldots, \mathsf{B}_k$, we first find out whether there is an immediate successor of A in the linear code layout. If so, the outgoing bindings of A is propagated to this successor. If there is no immediate successors, then we choose the successor connected via an edge with the highest execution frequency[6]. We have to insert compensation code along all other remaining edges.

[6]This can be obtained via profiling or some other static analysis.

3.6 Critical edge splitting

As noted before, critical edges that require compensation code have to be split. Splitting critical edges may degrade performance because extra branch instructions may have to be inserted into the program. The bindings propagation heuristic that we use will try to minimize this effect by splitting the critical edges with the lowest execution frequency.

Nevertheless, in order to further eliminate the effect of critical edge splitting, we try to share compensation code as much as possible. This is detected using the following heuristics.

1. Given a potential splitting candidate $\mathsf{A} \to \mathsf{B}$, if no floating point registers are live-out from A then no compensation code is necessary.

2. Suppose we are given critical edges $\mathsf{A}_1 \to \mathsf{B}, \ldots \mathsf{A}_k \to \mathsf{B}$ and the incoming bindings of B is empty, i.e., no floating point registers are live. Then the compensation code that we generate for each edge is simply the appropriate number of pops. This compensation code can be shared among all the critical edges. Furthermore, it can be placed immediately before the block B if B does not have a falls through predecessor block.

   For example, if the outgoing stack height of A_1 is 5 and that of A_2 is 3. Then we can generate the following shared compensation code:

   ```
   A1_B:  fstp %st(0)
          fstp %st(0)
   A2_B:  fstp %st(0)
          fstp %st(0)
          fstp %st(0)
   B:     ...
   ```

   The targets of blocks A_1 and A_2 can be redirected to the label `A1_B` and `A2_B` respectively.

3. In the general case, we are given the critical edges $\mathsf{A}_1 \to \mathsf{B}, \ldots \mathsf{A}_k \to \mathsf{B}$. We can share the compensation code between two edges $\mathsf{A}_i \to \mathsf{B}$ and $\mathsf{A}_j \to \mathsf{B}$ if blocks A_i and A_j have the same outgoing stack bindings.

4 Bugs

Single and 80-bit precision floating point has not been fully implemented. Also, the C-call interface code has not been updated.

References

[CH90] F. C. Chow and J. L. Hennessy. The priority–based coloring approach to register allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, 1990.

[Cor97] Intel Corporation. Intel architecture optimization manual. Technical Report 242816–003, Intel Corporation, 1997.

[DRZ92] D. Dhamdhere, B. Rosen, and K. Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, San Francisco, California, June 1992.

[GA96] L. George and A. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(5), May 1996.

[Geo99] L. George. Register allocation for architectures with few registers (`http://cm.bell-labs.com/cm/cs/what/smlnj/compiler-notes/k32.ps`). Technical report, Bell Laboratories, Murray Hill, 1999.

[LG99] A. Leung and L. George. A new mlrisc register allocator (`http://cm.bell-labs.com/cm/cs/what/smlnj/compiler-notes/new-ra.ps`). Technical report, Bell Laboratories, Murray Hill, 1999.

[LG00] A. Leung and L. George. Register allocation for architectures with few registers (in preparation). Technical report, New York University and Bell Laboratories, Murray Hill, 2000.

[PF91] T. A. Proebsting and C. N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 256–267, Toronto, Ontario, Canada, June 1991.

[SU70] R. Sethi and J. D. Ullman. The generation of optimal code for arthmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

A Register Parameter Passing

Livein and liveout sets at procedure entries and exits are interpreted specially as the `%st(0)` registers so that floating point register parameters can be specified.

For example, we can generate a floating point squaring function that passes its parameter and result in `%st(0)` as follows in MLTREE:

```
(* MLTREE *)
structure T = MLTree
structure C = Cells

val st0 = C.ST 0
...
   beginCluster 0;
   emit(T.FMV(st0, T.FMUL(64, T.FREG(64, st0), T.FREG(64, st0))));
   emit(T.RET);
   exitBlock[T.FPR st0]; (* %st(0) is live out *)
   endCluster []
```

B Specializing the MLRISC Register Allocator

Here is how we can specialize the MLRISC register allocator to use the new floating point code generator. We will use the SML/NJ backend as a running example. For basic documentation on how to instantiate the register allocator, please see [LG99].

The operand datatype of the x86 instruction set has been extended with a new constructor `FPR`.

```
datatype operand =
    ...
  | FDirect    of $FP
  | FPR        of $FP
```

Under the new mode, `FPR` stands for a floating point virtual register, while the constructor `FDirect` stands for a floating point pseudo memory register.

The instruction selection module takes a new parameter `fast_floating_point` which indicates whether fast floating point code should be generated. This is a boolean flag that can be turned on or off dynamically during compilation The two modes of compilation are entirely compatible if the calling conventions of places floating point values in memory rather than on the stack.

```
val fast_floating_point = MLRiscControl.getFlag "x86-fast-fp"

structure MLTreeComp=
   X86(structure X86Instr=X86Instr
       structure X86MLTree=X86MLTree
       ...
       val fast_floating_point = fast_floating_point
      )
```

To use the new floating point code generation, the new floating point rewriting phase have to be instantiated. This can be done as follows:

```
structure X86FP =
   X86FP(structure X86Instr = X86Instr
         structure X86Props = InsnProps
         structure Flowgraph = X86FlowGraph
         structure Liveness = Liveness(X86FlowGraph)
         structure Asm = X86AsmEmitter
        )
```

The X86 spilling module and the x86 register allocator are instantiated in the usual way. In the following, we use the Chow and Hennessy [CH90] spilling heuristics and enable the memory coalescing module.

```
structure X86Spill =
    X86Spill(structure Instr=I
```

```
              structure Props=InsnProps)

structure Ra =
  RegisterAllocator
   (ChowHennessySpillHeur)
   (MemoryRA             (* for memory coalescing *)
     (ClusterRA
        (structure Flowgraph = F
         structure Asm = X86AsmEmitter
         structure InsnProps = InsnProps
         structure Spill = RASpill
            (structure Asm = X86AsmEmitter
             structure InsnProps = InsnProps
            )
        )
     )
   )
```

The round robin coloring module is instantiated so that registers `%f0`, ..., `%f6` are available:

```
val availF8 = map I.C.FPReg [0,1,2,3,4,5,6]
val KF8  = length availF8
structure FR8  = GetReg(val nRegs=KF8
                        val available=availF8
                        val first=I.C.ST 0)
```

Next, we provide all the callbacks to the register allocator.
The following callback is used to reconstruct a parallel copy.

```
fun copyInstrF((rds as [_], rss as [_]), _) =
     [I.FCOPY{dst=rds, src=rss, tmp=NONE}]
  | copyInstrF((rds, rss), I.FCOPY{tmp, ...}) =
     [I.FCOPY{dst=rds, src=rss, tmp=tmp}]
```

The following routine is needed to compute the effective address of a spill location from a logical spill location number. Non-negative logical spill locations actually denote a pseudo memory register, so we return `I.FDirect loc`. Otherwise, we lookup the effective address that has been assigned to `loc`, or allocate a new one when none exists.

```
  fun getFregLoc loc =
      if loc >= 0 then I.FDirect loc
      else
        I.Displace{base=esp,
                   disp=<lookup location>, mem=spill}
```

Normally, we use the following callbacks for the floating register allocator.

```
fun spillF{instr, reg, spillLoc, kill, regmap, annotations} =
     X86Spill.fspill(instr, regmap, reg, getFregLoc spillLoc)

fun spillFreg{src, reg, spillLoc, annotations} =
    [I.FLDL(I.FDirect(src)), I.FSTPL(getFregLoc spillLoc)]

fun spillFcopyTmp{copy=I.FCOPY{dst, src, ...}, spillLoc, annotations} =
     I.FCOPY{dst=dst, src=src, tmp=SOME(getFregLoc spillLoc)}

(* rename floating point *)
fun renameF{instr, fromSrc, toSrc, regmap} =
     X86Spill.freload(instr, regmap, fromSrc, I.FDirect toSrc)

(* reload floating point *)
fun reloadF{instr, reg, spillLoc, regmap, annotations} =
     X86Spill.freload(instr, regmap, reg, getFregLoc spillLoc)

fun reloadFreg{dst, reg, spillLoc, annotations} =
     [I.FLDL(getFregLoc spillLoc), I.FSTPL(I.FDirect dst)]
```

For the new floating point mode, however, we will use the following callbacks instead. The main difference is that these callbacks generate pseudo floating point instructions.

```
fun FMemReg f = if f >= I.C.FPR 8 andalso f < I.C.FPR 32
                then I.FDirect f else I.FPR f

fun copyInstrF'((rds as [d], rss as [s]), _) =
     [I.FMOVE{fsize=I.FP64,src=FMemReg s,dst=FMemReg d}]
  | copyInstrF'((rds, rss), I.FCOPY{tmp, ...}) =
     [I.FCOPY{dst=rds, src=rss, tmp=tmp}]

fun spillFreg'{src, reg, spillLoc, annotations} =
     [I.FMOVE{fsize=I.FP64, src=FMemReg src, dst=getFregLoc spillLoc}]

fun renameF'{instr, fromSrc, toSrc, regmap} =
     X86Spill.freload(instr, regmap, fromSrc, I.FPR toSrc)

fun reloadFreg'{dst, reg, spillLoc, annotations} =
     [I.FMOVE{fsize=I.FP64, dst=FMemReg dst, src=getFregLoc spillLoc}]
```

The parameters to the register allocator are as follows. In this example, we are also enabling the memory coalescing capability of the register allocator. Pseudo memory registers are associated with the range `[I.C.FPReg 8,I.C.FPReg 31]`.

```
val RAInt = ... (* integer RA parameters *)
val RAFP8 =
```

```
{spill     = spillF,
 spillSrc  = spillFreg',
 spillCopyTmp= spillFcopyTmp,
 reload    = reloadF,
 reloadDst = reloadFreg',
 renameSrc = renameF',
 copyInstr = copyInstrF',
 K         = KF8,
 getreg    = FR8.getreg,
 cellkind  = I.C.FP,
 dedicated = dedicatedF8,
 spillProh = [],
 memRegs   = [(I.C.FPReg 8,I.C.FPReg 31)],
 mode      = Ra.SPILL_PROPAGATION+Ra.SPILL_COLORING
} : Ra.raClient
```

Finally, everything can be tied together:

```
fun ra cluster =
let val _ = spillInit()
    val cluster = Ra.ra [RAInt, RAFP8] cluster
    val cluster = X86FP.run cluster
in  cluster
end
```