cl-simd

This library implements SSE intrinsic functions for ECL and SBCL. It provides access to SSE2 instructions (which are nowadays supported by any CPU compatible with x86-64) in the form of *intrinsic functions*, similar to the way adopted by modern C compilers. It also provides some lisp-specific functionality, like setf-able intrinsics for accessing lisp arrays.

This API, with minor technical differences, is supported by both ECL and SBCL (x86-64 only).

When this module is loaded, it defines an `:sse2` feature, which can be subsequently used for conditional compilation of code that depends on it. Intrinsic functions are available from the `sse` package.

NOTE: CURRENTLY THIS SHOULD BE CONSIDERED EXPERIMENTAL, AND SUBJECT TO INCOMPATIBLE CHANGES IN A FUTURE RELEASE.

Since the implementation is closely tied to the internals of the compiler, it should normally be obtained exclusively via the bundled contrib mechanism of the above implementations.

SSE pack types

The package defines and/or exports the following types to represent 128-bit SSE register contents:

Package: ***sse*** The packages where the cl-simd symbols are present.

Type: ***sse-pack*** **&optional item-type** The generic SSE pack type.

Type: ***int-sse-pack*** Same as `(sse-pack integer)`.

Type: ***float-sse-pack*** Same as `(sse-pack single-float)`.

Type: ***double-sse-pack*** Same as `(sse-pack double-float)`.

Declaring variable types using the subtype appropriate for your data is likely to lead to more efficient code (especially on ECL). However, the compiler implicitly casts between any subtypes of sse-pack when needed.

Printed representation of SSE packs can be controlled by binding `*sse-pack-print-mode*`:

Variable: ***sse-pack-print-mode*** When set to one of `:int`, `:float` or `:double`, specifies the way SSE packs are printed. A `NIL` value (default) instructs the implementation to make its best effort to guess from the data and context.

SSE array type

Type: *sse-array* element-type &optional dimensions Expands to a lisp array type that is efficiently supported by AREF-like accessors. It should be assumed to be a subtype of `SIMPLE-ARRAY`. The type expander signals warnings or errors if it detects that the element-type argument value is inappropriate or unsafe.

Function: *make-sse-array* dimensions &key element-type initial-element displaced-to displaced-
Creates an object of type `sse-array`, or signals an error. In non-displaced case ensures alignment of the beginning of data to the 16-byte boundary. Unlike `make-array`, the element type defaults to (unsigned-byte 8).

On ECL this function supports full-featured displacement. On SBCL it has to simulate it by sharing the underlying data vector, and does not support nonzero index offset.

Differences from C intrinsics

Intel Compiler, GCC and MSVC[1] all support the same set of SSE intrinsics, originally designed by Intel. This package generally follows the naming scheme of the C version, with the following exceptions:

- Underscores are replaced with dashes, and the `_mm_` prefix is removed in favor of packages.
- The `e` from `epi` is dropped because MMX is obsolete and won't be supported.
- `_si128` functions are renamed to `-pi` for uniformity and brevity. The author has personally found this discrepancy in the original C intrinsics naming highly jarring.
- Comparisons are named using graphic characters, e.g. `<=-ps` for `cmpleps`, or `/>-ps` for `cmpngtps`. In some places the set of comparison functions is extended to cover the full possible range.
- Scalar comparison predicates are named like `..-ss?` for `comiss`, and `..-ssu?` for `ucomiss` wrappers.
- Conversion functions are renamed to `convert-*-to-*` and `truncate-*-to-*`.
- A few functions are completely renamed: `cpu-mxcsr` (setf-able), `cpu-pause`, `cpu-load-fence`, `cpu-store-fence`, `cpu-memory-fence`, `cpu-clflush`, `cpu-prefetch-*`.

[1] http://msdn.microsoft.com/en-us/library/y0dh78ez%28VS.80%29.aspx

In addition, foreign pointer access intrinsics have an additional optional integer offset parameter to allow more efficient coding of pointer deference, and the most common ones have been renamed and made SETF-able:

- `mem-ref-ss`, `mem-ref-ps`, `mem-ref-aps`
- `mem-ref-sd`, `mem-ref-pd`, `mem-ref-apd`
- `mem-ref-pi`, `mem-ref-api`, `mem-ref-si64`

(The `-ap*` version requires alignment.)

Comparisons and NaN handling

Floating-point arithmetic intrinsics have trivial IEEE semantics when given QNaN and SNaN arguments. Comparisons have more complex behavior, detailed in the following table:

Single-float	Double-float	Condition	Result for NaN	QNaN traps
`=-ss,=-ps`	`=-sd,=-pd`	Equal	False	No
`<-ss,<-ps`	`<-sd,<-pd`	Less	False	Yes
`<=-ss,<=-ps`	`<=-sd,<=-pd`	Less or equal	False	Yes
`>-ss,>-ps`	`>-sd,>-pd`	Greater	False	Yes
`>=-ss,>=-ps`	`>=-sd,>=-pd`	Greater or equal	False	Yes
`/=-ss,/=-ps`	`/=-sd,/=-pd`	Not equal	True	No
`/<-ss,/<-ps`	`/<-sd,/<-pd`	Not less	True	Yes
`/<=-ss,` `/<=-ps`	`/<=-sd,` `/<=-pd`	Not less or equal	True	Yes
`/>-ss,/>-ps`	`/>-sd,/>-pd`	Not greater	True	Yes
`/>=-ss,` `/>=-ps`	`/>=-sd,` `/>=-pd`	Not greater or equal	True	Yes
`cmpord-ss,` `cmpord-ps`	`cmpord-sd,` `cmpord-pd`	Ordered, i.e. no NaN args	False	No
`cmpunord-ss,` `cmpunord-ps`	`cmpunord-sd,` `cmpunord-pd`	Unordered, i.e. with NaN args	True	No

Likewise for scalar comparison predicates, i.e. functions that return the result of the comparison as a Lisp boolean instead of a bitmask sse-pack:

Single-float	Double-float	Condition	Result_for_NaN	QNaN_traps
`=-ss?`	`=-sd?`	Equal	True	Yes
`=-ssu?`	`=-sdu?`	Equal	True	No
`<-ss?`	`<-sd?`	Less	True	Yes
`<-ssu?`	`<-sdu?`	Less	True	No
`<=-ss?`	`<=-sd?`	Less_or_equal	True	Yes
`<=-ssu?`	`<=-sdu?`	Less_or_equal	True	No
`>-ss?`	`>-sd?`	Greater	False	Yes
`>-ssu?`	`>-sdu?`	Greater	False	No
`>=-ss?`	`>=-sd?`	Greater_or_equal	False	Yes
`>=-ssu?`	`>=-sdu?`	Greater_or_equal	False	No
`/=-ss?`	`/=-sd?`	Not_equal	False	Yes
`/=-ssu?`	`/=-sdu?`	Not_equal	False	No

Note that MSDN specifies different return values for the C counterparts of some of these functions when called with NaN arguments, but that seems to disagree with the actually generated code.

Simple extensions

This module extends the set of basic intrinsics with the following simple compound functions:

- `neg-ss`, `neg-ps`, `neg-sd`, `neg-pd`, `neg-pi8`, `neg-pi16`, `neg-pi32`, `neg-pi64`:

  implement numeric negation of the corresponding data type.

- `not-ps`, `not-pd`, `not-pi`:

  implement bitwise logical inversion.

- `if-ps`, `if-pd`, `if-pi`:

  perform element-wise combining of two values based on a boolean condition vector produced as a combination of comparison function results through bitwise logical functions.

The condition value must use all-zero bitmask for false, and all-one bitmask for true as a value for each logical vector element. The result is undefined if any other bit pattern is used.

N.B.: these are *functions*, so both branches of the conditional are always evaluated.

The module also provides symbol macros that expand into expressions producing certain constants in the most efficient way:

- 0.0-ps 0.0-pd 0-pi for zero
- true-ps true-pd true-pi for all 1 bitmask
- false-ps false-pd false-pi for all 0 bitmask (same as zero)

Lisp array accessors

In order to provide better integration with ordinary lisp code, this module implements a set of AREF-like memory accessors:

- `(ROW-MAJOR-)?AREF-PREFETCH-(T0|T1|T2|NTA)` for cache prefetch.
- `(ROW-MAJOR-)?AREF-CLFLUSH` for cache flush.
- `(ROW-MAJOR-)?AREF-[AS]?P[SDI]` for whole-pack read & write.
- `(ROW-MAJOR-)?AREF-S(S|D|I64)` for scalar read & write.

(Where A = aligned; S = aligned streamed write.)

These accessors can be used with any non-bit specialized array or vector, without restriction on the precise element type (although it should be declared at compile time to ensure generation of the fastest code).

Additional index bound checking is done to ensure that enough bytes of memory are accessible after the specified index.

As an exception, ROW-MAJOR-AREF-PREFETCH-* does not do any range checks at all, because the prefetch instructions are officially safe to use with bad addresses. The AREF-PREFETCH-* and *-CLFLUSH functions do only ordinary index checks without the usual 16-byte extension.

Example

This code processes several single-float arrays, storing either the value of a*b, or c/3.5 into result, depending on the sign of mode:

```
(loop for i from 0 below 128 by 4
   do (setf (aref-ps result i)
            (if-ps (<-ps (aref-ps mode i) 0.0-ps)
                   (mul-ps (aref-ps a i) (aref-ps b i))
                   (div-ps (aref-ps c i) (set1-ps 3.5)))))
```

As already noted above, both branches of the if are always evaluated.