CSC 402

Simple C like SIMD focused language

Research, Compiler, Specification

The purpose of this class:

The purpose of CSC 402 is to both investigate the SIMD capabilities present in modern hardware, and to design and write a language and compiler; ultimately acting as a stand in for the compilers class, which has not been taught for several years.

On the emergence and current support of SIMD:

Over the past 25 years the microprocessor industry has moved towards providing ‘short vector’ SIMD faculties; with Intel, ARM, and RISC-V all providing similar expansions.

Intel’s SIMD faculties began with the MMX integer SIMD extensions in 1997 on their new Pentiums, since then multiple complimentary extensions have been implemented (1). The SSE extensions (SSE, SSE2, SSE3, SSSE3, SSE4) have been largely focused on floating point operations but have continuously added new integer instructions on top of the old MMX set. Here in lies the first difficulty: integer SIMD orthogonality is spotty on the x86-64 platform. Basic instructions do not necessarily exist for all integer sizes. As an example, instructions such as shifting and multiplication are absent for byte vectors, and no integer division facilities are provided. Additionally, despite how the integer and floating-point instructions share a register file, there are domain change penalties. Meaning that more flexible floating-point shuffle and manipulation instructions may not be best to use with integer registers, and vice-versa (1).

This was mitigated with newer AVX instructions, but adoption is slow. There has also been a massive roadblock with the AVX-512 instructions. Very few CPUs implemented the full feature set, and the 512bit vectors proved incompatible with Intel’s Efficiency Core design, leading to the extension being retired (6). While the upcoming APX extension intends to return the instructions added in AVX-512 (AVX-10), it is without the 512bit vectors (7). As such, it was decided to only use instructions added in SSE4.2 and prior in the compiler.

The ARM NEON extensions provide similar facilities to the x86 SSE-type instructions (5). Featuring a register file consisting of thirty-two 64bit SIMD registers, which may be paired into sixteen 128bit registers. These registers are used for both integer and floating-point operations, but due to the embedded system focus of most ARM implementations, a floating-point unit is not guaranteed, and the instructions are generally less floating point focused; no support for double precision floating-point is provided. Unlike the base SSE instructions however, NEON provides a length mask register, which can mask the load, store, and operations of and on SIMD registers.

RISC-V supplies two types of vector extensions: both a traditional SIMD style (‘P’ extension), and a variable “Cray” vector style (‘V’ extension) (4). The “Cray” style refers to the Cray 1, which used special loop control registers and masks, such that a fixed length vector register may be made to function as a variable length one in a loop (8). By far the ‘V’ extension is the preferred one, and the ‘P’ extension is currently not final. The V extensions additionally provide Cray style gather and scatter operations, while a traditional SIMD design is unable to cope with non-sequential elements, instead treating the vector register as one large word. By far the Cray design is the most flexible, as abstracting the length of the vector register allows a loop written for a cheaper RISC-V processor with short vectors to run at an enhanced speed on a more expensive processor, with far longer vectors, without the need to rewrite the loop.

There remains a discussion about the benefits of SIMD style extensions versus Cray style. Short SIMD vectors have a relatively low overhead, and do not require both the scatter-gather units and masking operations that are indicative of the Cray style. Yet the Cray style is far more flexible with changing hardware. SIMD extensions can also be upgraded with Cray style facilities; the AVX series of extensions on x86-64 provide a scatter unit and improved operations, with AVX-512 adding both a gather unit, longer registers, and a vector mask system. The longer registers will not be carried over into AVX-10 however and raises the question if longer and larger vector register files are ideal on consumer hardware. The thirty-two 512bit ZMM registers in AVX-512 is a two-kilobyte monster that must be swapped in and out on every context switch (1) and produce so much heat they were not viable for performance cores (6). This contrasts with the very long registers of GPU systems, which do not have to worry about the logistics of quickly and repeatedly storing and loading multi-megabytes of register file.

Ultimately, the language and compiler will focus on the much more popular SIMD style, will feature the popular vector permute, vector extract, and vector reduce (vector length running sum) operations, and will be written to the SSE4.2 and previous extensions for simplicities sake.

On the language designed:

Simplified C-style with SIMD, or SC-SIMD, is a simplified C like toy language with a focus on one dimensional 'short' SIMD vectors, inspired by the current trends in consumer hardware. The C specification was used as a starting point, and a great deal was removed to simplify the task of writing the compiler. This includes structs, unions, most preprocessor features, a simplification of the type system, the simplification of reference operator "&", and the removal of array's syntactic sugar.

Reflection on the Compiler:

The compiler for this project implements the vast majority of the language, with the exception of vector permute, vector running sum, the preprocessor, variable function arguments "…", and any byte vector operations. It also currently restricts vector lengths to a total of no more than 128bits, while the specification allows up to 128 elements. It was written in C, using Flex and Bison for the lexer and parser, respectively. The compiler project is broken up into two parts, a front end that handles syntax checking, type checking, and common subexpression elimination. And converts it to a directed acyclic graph. The back end transforms the DAG directly into x86 GNU syntax assembler by traversing the graph largely depth first.

Originally, I wished to add a middle step, which turns the DAG into three address code, simplifies the operations and inlines functions. The back end would then turn this into valid assembler and performs peephole optimizations. Ultimately, I was unable to do this in the allotted time. While the compiler was ultimately able to produce working code for a demo file, the final project is far buggier and more difficult to read and maintain than I was wishing for. While I feel that I achieved the goal of learning how to write a compiler, the compiler written for the project itself leaves much to be desired. Over the next couple months, I intend to clean up and improve the source.

SC-SIMD Specification:

List of Base Type:

VOID	:	Non-type variable
BYTE	:	Single byte, eight bit, integer variable
WORD	:	Two byte, 16 bit, integer variable
LONG	:	Four byte, 32 bit, integer variable
INT	:	Alias for LONG
QUAD	:	Four word, eight byte, 64 bit, integer variable
INDEX	:	Same size as a pointer, Alias for QUAD
SINGLE	:	32bit floating point number
DOUBLE	:	64bit point number
< N >	:	Vector postfix of length N, from 2 to 128

An Explanation of Base Types:

Base types have been heavily simplified, and the types system is very strict. All integers are signed, and unsigned comparisons are to be done with the equation: "(signed) (A – B) < 0 ≡ (unsigned) (A > B)". Integers of differing sizes and floating points of different sizes cannot operate together. The compiler does not ensure that floating point operations are done in the order specified when performing redundant subexpression elimination, which will result in rounding errors. The base type names reflect their typing in GNU AS x86 assembler syntax. As there is no array indexing operation, only integers of the same size of a pointer may be used in pointer arithmetic. INDEX is provided for this purpose, which is guaranteed to be the same size as a pointer. Here, it is an alias for a QUAD. Vectors consist of a base type augmented by a length "< N >", where N is between 2 and 128. Only vectors of the same length and base type can operate on each other. A scalar cannot interact with a vector.

List of Type Modifiers:

...(parameter-list)	:	Marks function returning...
[...]	:	Marks pointer to...

An Explanation of Type Modifiers:

The function and pointer markers operate as they do in C, with the exception that they are no longer part of a variable's 'name', and are instead bound to the type. The Pointer marker and dereference operator "*foo" has been changed to "[foo]" so that indexing "[foo + bar]" is cleaner with the absence of first-class arrays.

List of Hint Modifiers:

GLOBAL...	:	Marks a declaration that extends outside the immediate file.
EXTERN...	:	On a partial declaration, marks an externally defined label.

An Explanation of Hint Modifiers:

HINT MODIFIERS trail left of the entire declaration; resulting in at most one per declaration. As they are simply hints to the compiler on how to set up the declaration in a possible symbol table. The compiler is to treat every definition defined in the base scope (not in a function) as if it was STATIC in C; marking something as GLOBAL actively tells the compiler not to do this. EXTERN functions as in C.

Constant (Numeric Literal) Conversions and Syntax:

- <opt> [0-9]+	:	decimal constant
0b [0-9A-Fa-f]+	:	binary constant
0x [0-9A-Fa-f]+	:	hex constant
[0-9]+. [0-9]+	:	floating point constant
"string"	:	character (.asciz) string constant
\ 0, 1, 2, 3 \	:	numeric string constant

An Explanation of Constant Conversions and Syntax:

All numeric constants are stored in the largest possible sign extended format, but are marked based on the smallest respective type that they may fit in. Once used in an operation with a typed variable, they are promoted, if need be, and are the only construct which can do so. Demotion results in an error. String constants return a constant pointer to the construct in the ASM file. Character Strings are null terminated.

Annotated Context Free Grammer Representation:

initial-expression

    IDENTIFIER
    CONSTANT
    ...
    (expression)
    & IDENTIFIER
    [expression]

Note:

‘…’ is the variable function parameter handle; implemented as a void pointer to the base of the stack frame. IDENTIFIERS are defined as [A-Fa-f][A-Fa-f0-9] with an optional leading ‘_’. The reference operator ‘&’ has been simplified to only operate directly on IDENTIFIERS. ‘[expression]’ is the dereference operator, as mentioned above.

postfix-operation

    initial-expression
    postfix-operation ( )
    postfix-operation (argument-list)
    postfix-operation < permute-list >
    postfix-operation < CONSTANT >

Note:

Permute and Pick (“exp <permute-list>” and “exp <constant>”) only operate on vectors. Function calls (“exp ()” and “exp (argument-list”)) operate on functions and function pointers.

permute-list

    CONSTANT
    permute-list , CONSTANT

Note:

Permute lists can only consist of constants and must match the length of the vector they are permuting. This operator stands for a series of SIMD shuffles.

argument-list

    ternary-operation
    argument-list , ternary-operation

prefix-operation

    postfix-operation
    ~ prefix-operation
    ! prefix-operation
    - prefix-operation
    + prefix-operation

Note:

Arithmetic negation “-“ and Logical negation “!” operate on scalers and vectors of all types. Bitwise negation “~” operates on scalers and vectors of only integers. Running sum “+” operates only on vectors of all base types.

multdiv-operation

    prefix-operation
    multdiv-operation * prefix-operation
    multdiv-operation / prefix-operation
    multdiv-operation % prefix-operation

Note:

Multiplication and Division operate on all types. Modulus only operates on integers.

addsub-operation

    multdiv-operation
    addsub-operation + multdiv-operation
    addsub-operation – multdiv-operation

Note:

Addition and Subtraction operate on all types.

shift-operation

    addsub-operation
    shift-operation >> addsub-operation
    shift-operation << addsub-operation

Note:

Left and Right shift operations only apply to integer types and are signed.

relation-operation

    shift-operation
    relation-operation < shift-operation
    relation-operation > shift-operation
    relation-operation <= shift-operation
    relation-operation >= shift-operation

Note:

All comparison operators operate on all types. The canonical FALSE value is 0, the canonical TRUE value is not 0. This applies across both integers and floating point.

equality-operation

    relation-operation
    equality-operation == relation-operation
    equality-operation != relation-operation

bitwise-and-operation

    equality-operation
    bitwise-and-operation & equality-operation

Note:

Bitwise operators only apply to integer types.

bitwise-eor-operation

    bitwise-and-operation
    bitwise-eor-operation ^ bitwise-and-operation

bitwise-or-operation

    bitwise-eor-operation
    bitwise-or-operation | bitwise-eor-operation

logical-and-operation

    bitwise-or-operation
    logical-and-operation && mesh-operation

Note:

Logical operates can mix integer and floating-point types, following the rule that 0 is FALSE and not 0 is TRUE. It cannot mix scalers and vectors however.

logical-or-operation

    logical-and-operation
    logical-or-operation || logical-and-operation

ternary-operation
    logical-or-operation
    logical-or-operation ? expression : ternary-operation

    Note:
        The ternary operator operates as a vector blend when all arguments are vectors, performing per lane swaps based on the canonical FALSE value 0.

expression
    ternary-operation

declaration
    type-name IDENTIFIER = expression ;
    type-name IDENTIFIER ;
    type-name IDENTIFIER scope

    Note:
        The empty declaration (type-name IDENT ;) functions both as a variable declaration and a function prototype, depending on the type.

hint-modifier
    EXTERN
    GLOBAL

    Note:
        These may only be applied outside of any level of scope nesting.

base-type
    VOID
    BYTE
    WORD
    LONG
    INT
    QUAD
    INDEX
    SINGLE
    DOUBLE

base-type-postfix
    base-type-prefix
    base-type-prefix < CONSTANT >

    Note: vector length CONSTANT must be between 2 and 128.

```
pointer-modifier
        [ type-base-postfix ]
        [ pointer-modifier  ]
        [ function-modifier ]

function-modifier
        type-base-postfix ( parameter-list )
        pointer-modifier  ( parameter-list )
        type-base-postfix ( parameter-list , ... )
        pointer-modifier  ( parameter-list , ... )
        type-base-postfix ( ... )
        pointer-modifier  ( ... )

        Note:
                "…" is the variable argument parameter.

parameter-list
        type-name IDENTIFIER
        parameter-list , type-name IDENTIFIER

type-name
        type-base-postfix
        function-modifier
        pointer-modifier
```

statement

```
        ;
        expression ;
        declaration
        scope
        IDENT = expression ;
        [ expression ] = expression ;
        IDENT < CONSTANT > = expression ;
        IF ( expression ) DO statement
        IF ( expression ) THAN statement ELSE statement
        WHILE ( expression ) statement
        BREAK ;
        CONTINUE ;
        RETURN expression<opt> ;
        IDENTIFIER : statement
        GOTO identifier ;
        FOR ( statement expression<opt> ; statement ) statement
```

Note:

IF, WHILE, and FOR may only accept scaler values for their “(expression)” arguments.

scope

```
        { statement
        scope statement
```

direcitive

```
        INCLUDE < FILENAME >
        INCLUDE " FILENAME "
        IFDEF IDENTIFIER THEN directive
        IFDEF IDENTIFIER THEN directive ELSE directive
        IFNDEF IDENTIFIER THEN directive
        IFNDEF IDENTIFIER THEN directive ELSE directive
        IF CONSTANT THEN directive
        IF CONSTANT THEN directive ELSE directive
        DEFINE IDENTIFIER
        UNDEF IDENTIFIER
        NOTHING
```

compiler-direcitive

```
        # directive \n
```

Annotated Bibliography:

(1) *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel, June 2021.

The Intel x86-64 programming guide. Details instructions and programming guides, along with the justification and intended use of several instructions. Used when writing the compiler, and selecting operations for the language.

(2) *ISO/IEC 9899:1999 (E),* Committee Draft, 1999.

The C99 ISO standard, was used as the basis for SC-SIMD's context free grammar. The C99 version was used as it is largely modern C, but is simpler than C11, C17, and C23.

(3) Aho, Alfred V. Sethi, Ravi. Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

Textbook detailing Compiler design and techniques. Outdated, but the information was still very pertinent. Advice on the use of Lex and Yacc was transferred to Flex and Bison, their GNU updates. Was referenced heavily throughout the project.

(4) *RISC-V specification,* RISC-V International, https://github.com/riscv. Accessed 12 Sep. 2023.

The complete RISC-V specification, hosted on GitHub. Also contains the justification and intended use of specific instructions and instruction groups. Referenced when selecting operations for the language.

(5) *Neon Programmers' Guide*, Documentation – Arm Developer, ARM Ltd., https://developer.arm.com/documentation/den0018/a/NEON-Instruction-Set-Architecture. Accessed 12 Sep. 2023.

Programmer's reference for ARM's NEON SIMD extension. Same as previous programmer's guides.

(6) Alcorn, Paul *Intel Nukes Alder Lake's AVX-512 Support, Now Fuses It off in Silicon,* Tom's Hardware, 2 Mar. 2022, www.tomshardware.com/news/intel-nukes-alder-lake-avx-512-now-fuses-it-off-in-silicon.

Article describing Intel's decision to disable, "fuse off", the AVX-512 instructions on their Alder Lake CPUs.

(7) Winkel, Sebastian. Agron, Jason. *Advanced Performance Extensions (APX),* Intel, www.intel.com/content/www/us/en/developer/articles/technical/advanced-performance-extensions-apx.html. Accessed 20 Oct. 2023.

Intel article outlining the upcoming APX/AVX-10 extensions, which reimplement many of the removed AVX-512 features.

(8) *The Cray-1 Computer Preliminary Reference Manual*, Cray Research, Inc. 1975
http://www.bitsavers.org/pdf/cray/CRAY-1/CRAY-1_PrelimRefRevA_Jun75.pdf

The Cray-1 Reference manual, acts as a programmer's reference. Provides the first example of a 'long vector' or 'true vector' computer.