SWARC

SIMD Within A Register C
Module Language & Compiler

Scc *versions from 061112*

Targets supported by **Scc***:*

- *Generic 32-bit C code*
- *MMX, 3DNow!, or SSE*
- *Altivec*
- *ATI DPVM CTM†*
- *OpenGL shaders†*
- *OpenGL+nVidia extensions†*

http://aggregate.org/SWAR/

Prof. Hank Dietz
Electrical and Computer Engineering Dept.
University of Kentucky
Lexington, KY 40506-0046
hankd@engr.uky.edu

SWARC, pronounced *swh-are-see*, is a C-like language designed to simplify writing portable code modules using SWAR (Simd Within A Register) parallelism. **Scc** is a module compiler for SWARC code. The language and compiler have been designed so that programmers can easily substitute SWARC code for appropriate functions within ordinary C programs, and ordinary C code can be used within SWARC code (e.g., to perform I/O operations).

Given SWARC's emphasis on efficiency, the typical execution cost for each language construct is indicated here by the sizes of □ for conventional processors and ○ for GPU targets: ▫◦ is fast parallel, □ ○ is somewhat parallel, and ☐ ◯ indicates slow/per-field operations. Things marked with † may not be fully implemented in the current version of **Scc**.

Data Types

SWARC Type	Meaning
char	C-layout 8-bit integer
short	C-layout 16-bit integer
int	C-layout 32-bit integer
float	C-layout 32-bit float
signed *type*	signed *type*
unsigned *type*	unsigned *type*
const *type*	read-only *type*
extern *type*	external/forward declaration
register *type*	register storage class
static *type*	static storage class
modular *type*	modular *type* (default)
saturation *type*	saturation version of *type*†
*type***:**	SWAR-layout *type*
*type***:***prec*	SWAR-layout *type*, *prec* bits
*type***[***width***]**	array of *width* values
typeof(*expr***)**	same type as *expr*

Notes: **char**, **short**, and **int** types with the same explicit precision are equivalent. *prec* and *width* can be compile-time constant expressions; actual precision is >=*prec*, but appears ==*prec* for **saturation**. Arrays can have only one dimension.

Type Coercion Rules:
1. For mixed widths, a width=1 (scalar) object is widened. For mixed widths>1, a warning is generated and the wider object is truncated.
2. Mixed C-layout and SWAR-layout yields the SWAR-layout & precision, even if precision is reduced.
3. Mixed precision yields higher precision.
4. Mixed signed and unsigned yields signed.
5. Mixed modular and saturation yields saturation.

E.g., mixing **signed int:2[20]** and **unsigned char:** yields **signed int:8[20]**; mixing that with **signed int[100]** yields **signed int:8[100]**.

Statements

SWARC statements implement "SIMD enable masking" for parallel operations. All functions begin with all elements enabled; **if**, **where**, **everywhere**, **while**, and **for** can change the enable set.

{ *block* **}**
▫◦ as in C; *block* of declarations & statements

${** *C_code* **$}
▫◦ allows arbitrary C code wherever a *stat* could appear. Within *C_code*, the **$** character is used to represent **#** so that nested C preprocessor runs can be used; e.g., **$include "file.h"** would include **file.h** in the C code at C-compile time

*label***:** *stat*
▫○ as in C; used with **goto** *label***;**

if (*expr***)** *stat* **else** *stat'*
□ ○ if *expr* has width==1, as in C; for width>1, the *stat* code is executed iff some enabled element is non-0, the *stat'* code is executed iff some enabled element is 0

where (*expr***)** *stat* **elsewhere** *stat'*
□ ◦ enable masking like **if**, but *stat* and *stat'* are always executed

everywhere *stat*
▫◦ enable all elements so that *stat* is executed without masking overhead

while (*expr***)** *stat*
□ ○ if *expr* has width==1, as in C; for width>1, the *stat* code is executed while at least one enabled element is non-0

for (*expr***;***expr***;***expr***)** *stat*
□ ○ as in C, same semantics as **while**

do *stat* **while (***expr***)**
▫ ○ if *expr* has width==1, as in C; if *expr* has width>1, the enable mask is unaffected, repeating *stat* while an enabled element is non-0

continue *expr***;**
▫○ as in C, extended to allow *expr* nesting levels

break *expr***;**
▫○ as in C, extended to allow *expr* nesting levels

return;
▫ ○ as in C, but SWARC only allows functions to return **void**

*ident***(***args...***);**
▫ ○ as in C; call a C or SWARC function *ident*, returning **void**

*expr***;**
▫◦ as in C

;
▫◦ as in C

Operators (precedence order)

expr assignment_op expr
extends C operator set and performs associative reductions (with masking) when storing width>1 value into width==1 variable; cost is □ ○ for **=**, **&&=**, **||=**, **?>=**, **?<=**, **+/=**, **-=**, ***=**, **/=**, **%=**; cost is □ ◯ for **>>=**, **<<=**, **&=**, **^=**, and **|=**

expr **?** *expr* **:** *expr*
▫◦ as in C, may use masking/arithmetic nulling

expr **||** *expr*

expr **&&** *expr*
▫◦ as in C, but yields 0 or -1

expr **|** *expr*

expr **^** *expr*

expr **&** *expr*
▫ ◯ as in C

expr equal_op expr
▫◦ as in C, but yields 0 or -1; operators are: **==** and **!=**

expr compare_op expr
▫◦ as in C, but yields 0 or -1 on simple compares: **<**, **>**, **<=**, and **>=**; extends C with minimum, maximum, and average operators: **?<**, **?>**, and **+/**

expr shift_op expr
▫ ◯ as in C; operators are: **>>** and **<<**

expr add_op expr
▫◦ as in C; operators are: **+** and **-**

expr mul_op expr
▫◦ as in C; operators are: *****, **/**, and **%**

prefix_op expr
as in C: ▫◦ for **-**, **++**, **--**, and **sizeof**; ▫ ◯ for **~**; like C, but yielding 0 or -1 for integer masking: ▫ ◦ **!**; extending C (including C*-like reductions): ▫ ◦ **widthof**, **precisionof**, **&&=**, **||=**, **?>=**, **?<=**, **+/=**, **+=**, **&=**, ***=**, **|=**, and **^=**

expr suffix_op
□ ○ array shift/rotate by constant *expr* operations: **[<<***expr***]**, **[<<%***expr***]**, **[>>***expr***]**, and **[>>%***expr***]**

Compile-Time Constants

widthof(*expr***)**
Width of *expr*, maximum data parallelism

precisionof(*expr***)**
Precision, in bits per element, of *expr*

Include files

#include "swarc.h"
SWARC equivalent to **stdio.h**

Suggested Development Procedure

Because SWARC is designed to be processed by a module compiler and linked to C routines, you probably will not develop codes using SWARC. The recommended development procedure is:

1. Develop your complete program as pure, portable, C code complying with the ANSI C specification (with **gcc** extensions permitted).
2. Benchmark your compiled C code. Unix tools like **gprof** are particularly useful in determining which functions dominate the execution time.
3. Multimedia instruction sets need very little data parallelism to achieve optimal speedup, no more than 512 bits per array; GPU targets need much longer vectors. If any of the functions identified in step 2 can use the appropriate flavor of parallelism, rewrite them as SWARC code. Where possible, use SWAR-layout data; this allows the compiler to use storage formats that are much more efficient, e.g., alignment/packing and storage in GPU texture memory.
4. Not all of the functions you rewrote will achieve speedup over the serial C code. Use **Scc**'s **-p** option to obtain detailed performance estimates.
5. Insert the SWARC functions in your C code. The programmer must ensure that the SWARC-generated code will be run only on hardware supporting the special instructions or GPU code generated; SWARC compilers do **not** automatically generate code to check that the correct hardware is present at runtime.

Note that, although SWARC code is somewhat portable and complexity (shown by □ and ○) of most operations is consistent across most targets, the precise speedups are machine dependent.

Sample Program

The following sample program defines a SWARC function called **addfour** that takes a C-layout first-class array of 2 integers (passed by address) and adds 4 to each of the elements. The **main** function is C code, defined inline within this SWARC program:

```
void addfour(int[2]x) { x += 4; }

${
main()
{
  int a[2]; a[0] = 1; a[1] = 2;
  addfour(a);
  printf("a={%d,%d}\n", a[0], a[1]);
}
$}
```

Compiled by the reference SWARC compiler using **Scc -Sc -P**, the header file **Scout.h** is something like:

```
extern void addfour(int *x);
```

For the default generic MMX target, the C code generated in **Scout.c** is something like:

```
#include "Sc.h"
void addfour(int *x)
{
  extern mmx_t mmx_cpool[];
  register mmx_t *_cpool = &(mmx_cpool[0]);
  {
    movq_m2r(*(((mmx_t *) x) + 0), mm0);
    paddd_m2r(*(_cpool + 0), mm0);
    movq_r2m(mm0, *(((mmx_t *) x) + 0));
  }
_return:
  emms();
}

main()
{
  int a[2]; a[0] = 1; a[1] = 2;
  addfour(a);
  printf("a={%d,%d}\n", a[0], a[1]);
}

/* MMX constant pool */
mmx_t mmx_cpool[] = {
/* 0 */ 0x0000000400000004LL
};
```

The actual assembly language translation of the program, as generated by **Scc test.Sc -S -O2**, includes code for **addfour** that looks like:

```
addfour:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%edx
  movl $mmx_cpool,%eax
#APP
  movq (%edx), %mm0
  paddd (%eax), %mm0
  movq %mm0, (%edx)
  emms
#NO_APP
  leave
  ret
```

Note that this final code incurs no additional overhead from use of inline assembly macros in **Scout.c**.