

Extending GCC with new operations: RTL, SIMD and treecodes

Zbigniew Chamski

Philips Electronic Design and Tools

`zbigniew.chamski@philips.com`

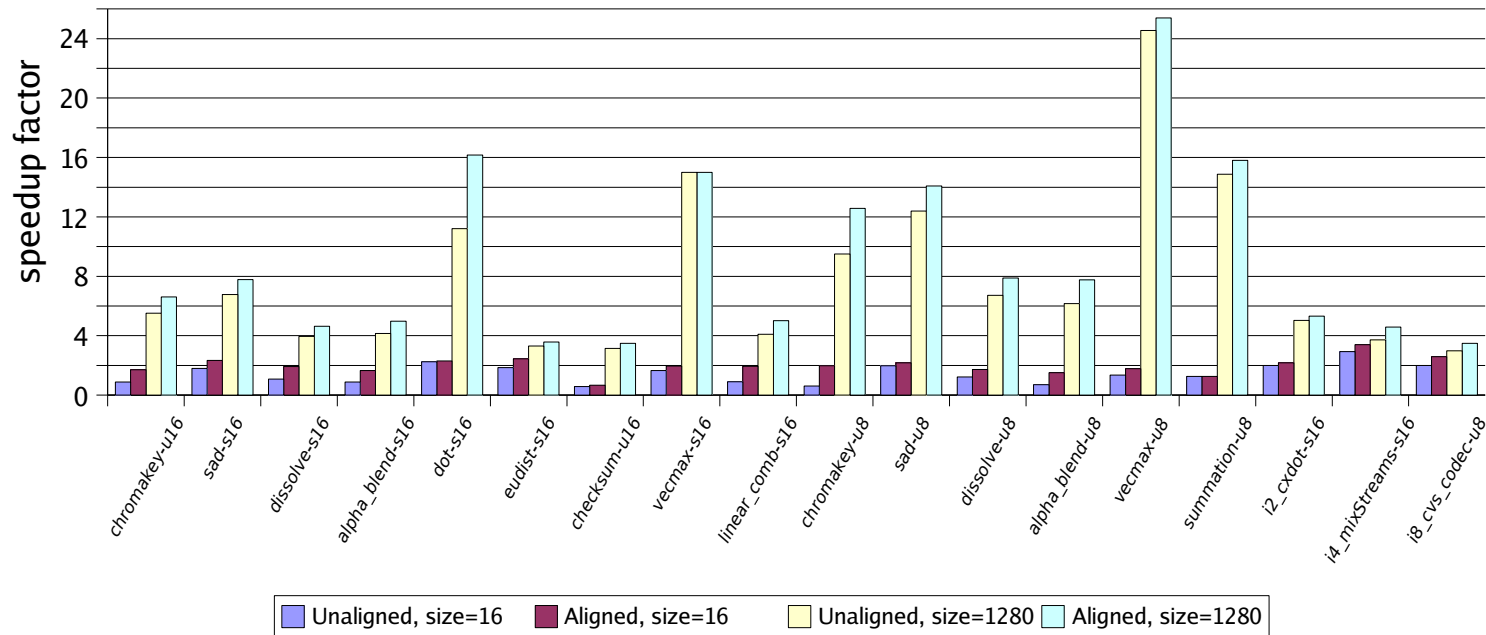
In this part

- Extending GCC? Why?
- Key elements of target support
- Adding new machine operations
- Adding SIMD vectorisation support for a target
- Adding new “core” operations: a brand new tree idiom
- Going forward...

A little motivation

mature target support + SIMD + new tree operations =

Performance of vectorized code wrt. sequential code, PowerPC970



gcc 4.2.0 autovect-branch, -O2 with/without vectorisation

Dorit Nuzman et al, *Automatic Vectorization for Embedded Targets*, submitted to Transactions on HiPEAC

Extendability mechanisms in GCC

- multi-targetability
 - clear separation between common and target-specific code
 - RTL: target-specific implementations of known idioms
 - trees: managing target- and language-independent concepts
- genericity
 - scalar and vector ops differ only in operand modes
 - contents of machine description determines which features (modes, operators) are available
- language-specific front-ends
 - C++, Fortran, Java, Ada...

Finding your way in the GCC source tree

- target-independent files are in `gcc/*.{c,h,def}`

File	Purpose
<code>gcc/tree.def</code>	definition of known tree-level idioms
<code>gcc/rtl.def</code>	definition of known RTL operators
<code>gcc/optabs.h</code>	declaration of operator tables for tree-to-RTL translations
<code>gcc/optabs.c</code>	definition of operator tables and intrinsics, basic support functions
<code>gcc/fold-const.c</code>	tree folding routines
<code>gcc/expr.c</code>	expansion of trees into RTL expressions

- target-specific support is in `gcc/config/<target_family>`
 - sufficiently powerful to support a new target
 - size: variable, few KLOC to few tens of KLOC

Key target-specific files

- machine description: *<target>.md*
 - definition of RTL instructions and their translations to assembly
- target-specific compiler options: *<target>.opt*
 - command-line options of GCC specific to the target
- target-specific definitions: *<target>.h*
 - basic parameters and features
- target-specific support functions: *<target>.c*
 - target predicates, code generation functions, target variants
- makefile fragments: *t-<target>*
 - special features of the build, e.g., multiple versions of libgcc

“What's in a name?” or the meaning of codes

- tree codes
 - indicate the nature of a tree, for building/traversal
 - type as a separate property of nodes
- optabs
 - tables of RTL operations sharing common semantics, but differing in operand size and/or structure
 - no type information available anymore
- RTL expression (“rtx”) codes
 - identify the purpose and structure of an RTL expression

The lifecycle of an operator in GCC

- detection/construction
 - parser, expression folding
- optimization
 - GIMPLE/SSA optimisations
- vectorization
 - some operators only make sense in loops: e.g., sum-of-abs-diffs
- expansion
 - convert trees into RTL expressions
- RTL optimisations, reload, and peephole optimisations

Target-specific implementation of existing idioms

- no need to alter the core of the compiler
 - it's all target-specific!
- simplest solution: MD file entries
 - applicable when the mappings are “simple”
 - additional knowledge can be expressed in plain C code
- let's look at a few examples
 - simple instructions
 - expansions to a sequence of instructions
 - folding multiple patterns into one

Anatomy of a simple RTL entry

signed MIN on Tensilica Xtensa (gcc/config/xtensa/xtensa.md)

```
(define_insn "sminsi3"
  [(set (match_operand:SI 0 "register_operand" "=a")
        (smin:SI (match_operand:SI 1 "register_operand" "%r")
                 (match_operand:SI 2 "register_operand" "r")))]
  "TARGET_MINMAX"
  "min\t%0, %1, %2"
  [(set_attr "type" "arith")
   (set_attr "mode" "SI")
   (set_attr "length" "3")])
```

unsigned MIN: the magic is in the operator, not in the modes

- replace “**smin**” by “**umin**”
- replace “**min**” by “**minu**”

RTL expansions: imperative coding

subtraction of wide integers [Tensilica Xtensa]

```

(define_expand "subdi3"
  [(set (match_operand:DI 0 "register_operand" "")
        (minus:DI (match_operand:DI 1 "register_operand" "")
                  (match_operand:DI 2 "register_operand" "")))]
  ""
  {
    rtx dstlo  = gen_lowpart (SImode, operands[0]);
    rtx src1lo = gen_lowpart (SImode, operands[1]);
    rtx src2lo = gen_lowpart (SImode, operands[2]);

    rtx dsthi  = gen_highpart (SImode, operands[0]);
    rtx src1hi = gen_highpart (SImode, operands[1]);
    rtx src2hi = gen_highpart (SImode, operands[2]);

    emit_insn (gen_subsi3 (dsthi, src1hi, src2hi));
    emit_insn (gen_subdi_carry (dsthi, src1lo, src2lo));
    emit_insn (gen_subsi3 (dstlo, src1lo, src2lo));

    DONE;
  })

```

Combining multiple patterns into one

- 1) if operands differ in nature - memory vs. register, value range of immediates, etc... - use multiple alternatives

[source: Xtensa port]

```
(define_insn "zero_extend_hisi2"
  [(set (match_operand:SI 0 "register_operand" "=a,a")
        (zero_extend:SI (match_operand:HI 1 "nonimmed_operand" "r,U")))]
  ""
  "@
  extui\t%0, %1, 0, 16
  l16ui\t%0, %1"
  [(set_attr "type" "arith,load")
   (set_attr "mode" "SI")
   (set_attr "length" "3,3")])
```

Constraint letters are handled in macros 'REG_CLASS_FROM_LETTER',
'CONST_OK_FOR_LETTER_P' and 'CONST_DOUBLE_OK_FOR_LETTER_P'

Combining multiple patterns into one

2) when the same pattern applies to multiple modes

- use *mode macros* to generate an entire family of patterns
- example: AltiVec SIMD support in gcc/config/rs6000/altivec.md

```
(define_mode_macro VI [V4SI V8HI V16QI])
(define_mode_attr VI_char [(V4SI "w") (V8HI "h") (V16QI "b")])
.....
(define_insn "altivec_vadds<VI_char>s"
  [(set (match_operand:VI 0 "register_operand" "=v")
        (unspec:VI [(match_operand:VI 1 "register_operand" "v")
                    (match_operand:VI 2 "register_operand" "v")]
                    UNSPEC_VADDS))
        (set (reg:SI 110) (unspec:SI [(const_int 0)] UNSPEC_SET_VSCR)))]
  "TARGET_ALTIVEC"
  "vadds<VI_char>s %0,%1,%2"
  [(set_attr "type" "vecsimple")])
```

When RTL patterns are not enough...

... you have to type a little C: [taken from AltiVec]

```
(define_insn "*mov<mode>_internal"
  [(set (match_operand:V 0 "nonimmediate_operand" "=Z,v,v,o,r,r,v")
        (match_operand:V 1 "input_operand" "v,Z,v,r,o,r,W"))]
  "TARGET_ALTIVEC
  && (register_operand (operands[0], <MODE>mode)
      || register_operand (operands[1], <MODE>mode))"
  {
    switch (which_alternative)
    {
      case 0: return "stvx %1,%y0";
      case 1: return "lvx %0,%y1";
      case 2: return "vor %0,%1,%1";
      case 3: return "#";
      case 4: return "#";
      case 5: return "#";
      case 6: return output_vec_const_move (operands);
      default: gcc_unreachable ();
    }
  }
  [(set_attr "type" "vecstore,vecload,vecsimple,store,load,*,*")])
```

More imperative coding: *<target>.c*

- complex move operations
 - example: `rs6000_split_multireg_move()`
- endianness-dependent behaviour
 - `if (BYTES_BIG_ENDIAN)`
- support of target variants
 - restrictions to the operation set, addressing modes, endianness
- manipulation of data representations
 - differences between host and target in cross-compilation
- generation of function prologue/epilogue code

Adding SIMD vectorisation support

- distinction between scalar and vector ops: **operand modes**
- availability of vector ops: deduced from MD file
- specify supported vector length: **<target>.h**

```
#define UNITS_PER_SIMD_WORD max_SIMD_bytes
```

- specify supported vector modes: **<target>-modes.def**

```
/* Vector modes.  */
VECTOR_MODES (INT, 8);           /*      V8QI V4HI V2SI */
VECTOR_MODES (INT, 16);        /* V16QI V8HI V4SI V2DI */
VECTOR_MODE (INT, DI, 1);
VECTOR_MODES (FLOAT, 8);       /*      V4HF V2SF */
VECTOR_MODES (FLOAT, 16);     /*      V8HF V4SF V2DF */
```

- build the compiler and invoke it with `-O2 -ftree-vectorize`

The life of an existing idiom: **MAX**

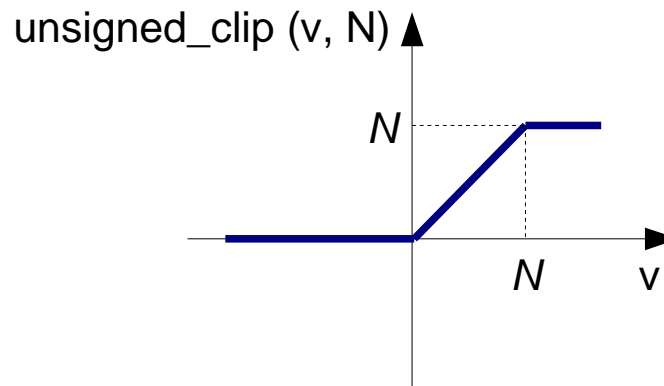
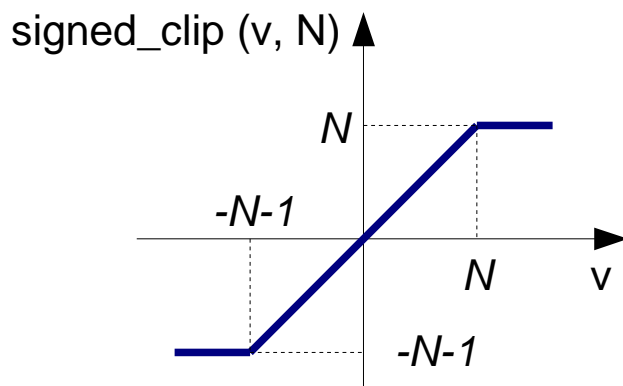
- source code pattern: `((a) > (b) ? (a) : (b))`
- treecode: `MAX_EXPR`
- two origins
 - detection of the pattern during expression folding, cf. `fold_cond_expr_with_comparison()` in `fold-const.c`
 - explicit invocation of the corresponding intrinsic operation
- availability on the target architecture
 - determined by the presence of appropriate RTL patterns in the MDF
 - presence of patterns is known to the core of the compiler
- RTL generation
 - fallback scenarios needed: detection done without looking at target

Let's add a brand new idiom to GCC

- **CLIP** (v , low , $high$):

$(v < low ? low : (v > high ? high : v))$
same as $MAX(low, MIN(v, high))$
same as $MIN(high, MAX(v, low))$

- purpose: ensure a value is in a specified range
- important in DSP: 250+ static occurrences in Xvid, 220+ in FFMPEG
- HW support: implicit lower bound, depending on result type



Adding a new idiom: general design choices

- arity
 - how much information must be provided explicitly?
- target-specific implementation or core operator?
- signedness handling
 - use sign of result to determine signedness of operation, vs. explicit signed/unsigned tree or operation code
- FP safety
 - NaNs, signed zeroes, Re/Im
- detection phase: operator folding? vectoriser?
- expansion to RTL: insn, expand, or C coding?

CLIP design considerations

- operator support in the target architecture
 - HW prefers symmetry, fewer operands and separate opcodes
 - decision: use implicit lower bound
 - consequence: use a binary operator
- signedness selection: at RTL generation time
 - a single tree code for signed/unsigned
 - selection of optab by result type
- invariant: signed result \Leftrightarrow signed CLIP operator
- expansion to RTL
 - native op if available, fall back on $\text{MIN}(\text{MAX}(\dots), \dots)$ combination

CLIP detection: what? and when?

- objective

fold *(v OP1 low ? low : (v OP2 high ? high : v))*
 into **CLIP_EXPR** *(v, low, high)*

- when?

- expression folding (**fold-const.c**): already during parsing
- repeated when new trees are constructed (**fold_buildN()** functions)

- folding done depth-first

- let's try the “minimum change” principle

- outer tree: “conditional expression with comparison”
- relevant existing code: “**fold_cond_expr_with_comparison()**”

Matching the tree pattern

- leverage already computed properties
 - `fold_cond_expr_with_comparison` always called on `(A op B ? A : C)`
 - `MIN` and `MAX` folding already took place ==> look for

`A op B ? A : MIN_or_MAX (B,C)`

- `MIN` and `MAX` are commutative: try permutations of operands!
- enforce the invariants: fold only if either
 - result is signed and the range is `-N-1 .. N`
 - result is unsigned and lower bound is 0

Not doing so may fool the RTL generation!

Changes to key files: **.def**

- **tree.def**: define the tree code

```
/* Clip arg1 to a signed (resp. unsigned) range, inclusive.
   Signed range is [ -arg2 - 1 .. arg2 ], whereas the unsigned range is [ 0 .. arg2 ].
   The signedness of the operation is determined by the type of the result. */
```

```
DEFTREECODE (CLIP_EXPR, "clip_expr", tcc_binary, 2)
```

- **rtl.def**: define the RTL expressions

```
/* Clip opd0 to a signed (resp. unsigned) range, inclusive.
   Signed range is [ -opd1 - 1 .. opd1 ], whereas the unsigned range is [ 0 .. opd1 ].
   Neither flavour of the operator is commutative, so the RTXes belong to the
   RTX_BIN_ARITH class, rather than to RTX_COMM_ARITH.
   */
```

```
DEF_RTL_EXPR(SCLIP, "sclip", "ee", RTX_BIN_ARITH)
```

```
DEF_RTL_EXPR(UCLIP, "uclip", "ee", RTX_BIN_ARITH)
```

Changes to key files: `optabs.h`

- add new operator tables to `enum optab_index`

```
/* Signed clip to range [-N-1 .. N] */  
OTI_sclip,  
/* Unsigned clip to range [zero .. N] */  
OTI_uclip,
```

- #define matching shortcuts

```
#define sclip_optab (optab_table[OTI_sclip])  
#define uclip_optab (optab_table[OTI_uclip])
```

Changes to key files: `optabs.c`

- add selection of appropriate optab in the dispatch function `optab_for_tree_code()`:

```
case CLIP_EXPR:
    return TYPE_UNSIGNED (type) ? uclip_optab : sclip_optab;
```

- initialise the new optabs in `init_optabs()`

```
sclip_optab = init_optab (SCLIP);
uclip_optab = init_optab (UCLIP);
```

- initialise information on corresponding intrinsic functions

```
init_integral_libfuncs (sclip_optab, "clip", '3');
init_integral_libfuncs (uclip_optab, "uclip", '3');
```

This will define intrinsic functions `__clipsi3`, `__uclipsi3` etc. for all modes for which `sclip/uclip` insns are defined in the MDF

Changes to key files, cont'd.

- do not forget to actually fill in the optabs! Add

```
"sclip_optab->handlers[$A].insn_code = CODE_FOR_$(sclip$a3$)",
"uclip_optab->handlers[$A].insn_code = CODE_FOR_$(uclip$a3$)",
```

to `static const char * const optabs[]` in **genopinit.c**

- provide default cases for the new treecode where needed:
 - rtx-to-tree code translation (`rtx_to_tree_code()` in `exprow.c`)
 - error reporting (`binary_op_error()` in `c-common.c`)
 - tree inliner (`estimate_num_insns_1()` in `tree-inliner.c`)
 - tree pretty-printer (`dump_generic_node()` in `tree-pretty-print.c`)
 - RTL optimiser (`simplify_binary_operation_1()` in `simplify_rtx.c`)
 - ...

Tree-to-RTL translation: `expr.c`

- expand to native operation if available

case **CLIP_EXPR**:

[... target location setup and operand expansion ...]

this_optab = optab_for_tree_code (code, type);

**temp = expand_binop (mode, this_optab, op0, op1, target, unsignedp,
OPTAB_WIDEN);**

if (temp != 0)

return temp;

[... fallback code ...]

- fallback scenario: expand a “lowered” **MAX(MIN())** tree

`uclip (v, high) ≡ MAX (0, MIN (high, v))`

`sclip (v, high) ≡ MAX (-high-1, MIN (high, v))`

- **MIN** and **MAX** expansions exist already... with fallbacks
- caveat: efficient **MAX (MIN ())** expansion may require aggressive CSE in RTL

RTL instruction definitions for CLIP

explicit unsigned CLIP [TriMedia, port under development]

```
(define_insn "uclipsi3"
  [(set (match_operand:SI 0 "register_operand" "=r")
        (uclip:SI (match_operand:SI 1 "reg_imm_operand" "r")
                  (match_operand:SI 2 "reg_imm_operand" "r")))]
  ""
  "uclipi %1 %2 -> %0"
  [(set_attr "type" "dspalu")])
```

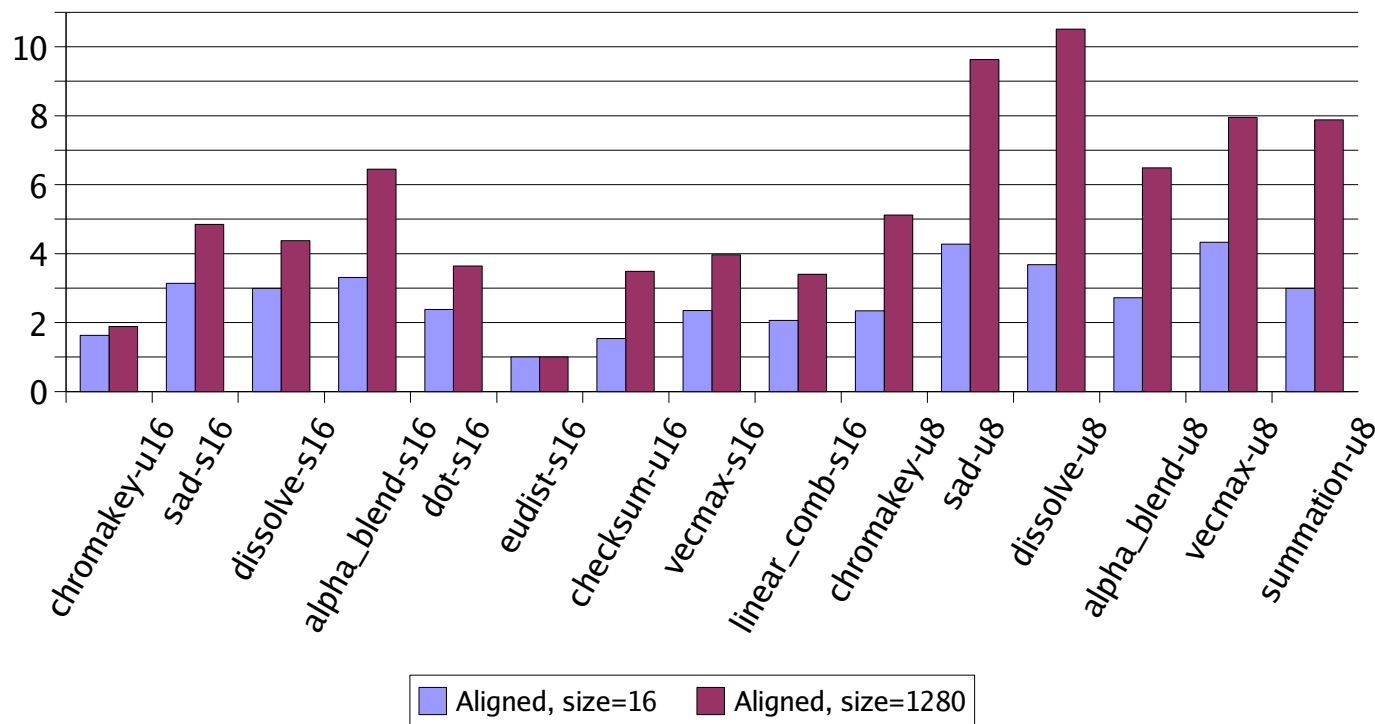
the same, for vectors of two half-precision integers

```
(define_insn "uclipv2hi3"
  [(set (match_operand:V2HI 0 "register_operand" "=r")
        (uclip:V2HI (match_operand:V2HI 1 "reg_imm_operand" "r")
                    (match_operand:V2HI 2 "reg_imm_operand" "r")))]
  ""
  "dualuclipi %1 %2 -> %0"
  [(set_attr "type" "dspalu")])
```

Prospects

new port, SIMD vectorization [TriMedia, under development]

Performance of vectorised code wrt. sequential code
(TriMedia TM3270)



source: Nuzman et al, *Automatic Vectorization for Embedded Targets*

If you plan to start a new port...

- first, check if there is
 - ongoing work (check <http://gcc.gnu.org> for announcements!)
 - interest: ask the community
- if the architecture is not public
 - you can always ask specific questions to the GCC community
- where to start?
 - check a recent port for current practice in MD coding
 - have a look at ports to targets “similar” to yours
 - it's OPEN source - do pattern matching:-)

Take-home message

- key knowledge: knowing where and what to add
- support for new ISAs (and/or variants thereof) is not that hard!
- SIMD support for a new target: an incremental change
- new ports: it's all about pattern matching and tree rewriting...