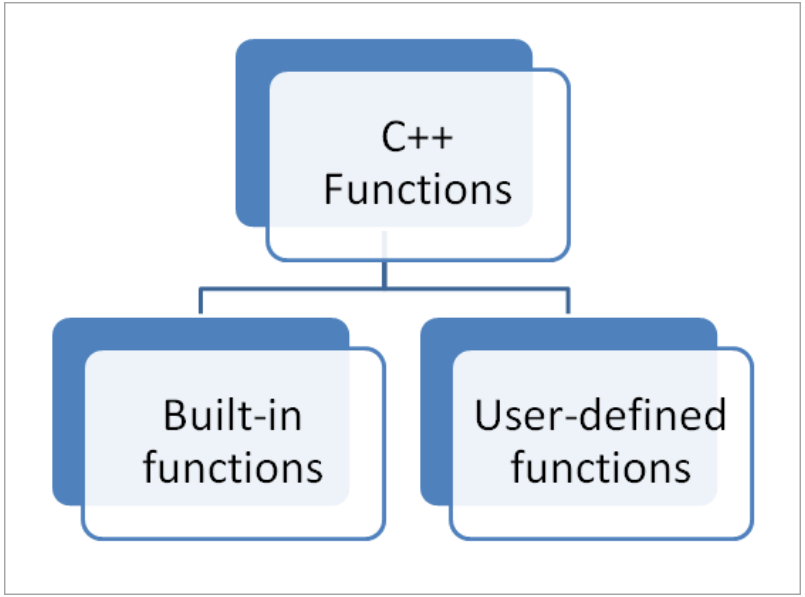
EMBECOSM®

How to add a GCC built-in to the RISC-V compiler

Nandni Jamnadas

What is a “built-in”

- Looks like a regular C function
- Intrinsic to the compiler, implemented directly within
- Patterns to be matched in the machine description
- Access to unique individual machine functionality
- For RISC-V, this presents an excellent opportunity to expose ISA extension functionalities



What is a “built-in”

```
/* no need to include math.h! */
void foo(void) {
    float two = __builtin_sqrtf(4.0);
}
```

- Not a real function
  - no entry or exit
  - only call as a function, cannot take the address

What is a “built-in”

- Standard builtins in gcc/builtins.def:

```
DEF_C99_C90RES_BUILTIN (BUILT_IN_SINF, "sinf", BT_FN_FLOAT_FLOAT, ATTR_MATHFN_FPROUNDING)
DEF_LIB_BUILTIN (BUILT_IN_SINH, "sinh", BT_FN_DOUBLE_DOUBLE, ATTR_MATHFN_FPROUNDING_ERRNO)
DEF_C99_C90RES_BUILTIN (BUILT_IN_SINHF, "sinhf", BT_FN_FLOAT_FLOAT, ATTR_MATHFN_FPROUNDING_ERRNO)
DEF_C99_C90RES_BUILTIN (BUILT_IN_SINHL, "sinhl", BT_FN_LONGDOUBLE_LONGDOUBLE, ATTR_MATHFN_FPROUNDING_ERRNO)
DEF_C99_C90RES_BUILTIN (BUILT_IN_SINL, "sinl", BT_FN_LONGDOUBLE_LONGDOUBLE, ATTR_MATHFN_FPROUNDING)
DEF_LIB_BUILTIN (BUILT_IN_SQRT, "sqrt", BT_FN_DOUBLE_DOUBLE, ATTR_MATHFN_FPROUNDING_ERRNO)
DEF_C99_C90RES_BUILTIN (BUILT_IN_SQRTF, "sqrtf", BT_FN_FLOAT_FLOAT, ATTR_MATHFN_FPROUNDING_ERRNO)
DEF_C99_C90RES_BUILTIN (BUILT_IN_SQRTL, "sqrtl", BT_FN_LONGDOUBLE_LONGDOUBLE, ATTR_MATHFN_FPROUNDING_ERRNO)
#define SQRT_TYPE(F) BT_FN_##F##_##F
DEF_EXT_LIB_FLOATN_NX_BUILTINS (BUILT_IN_SQRT, "sqrt", SQRT_TYPE, ATTR_MATHFN_FPROUNDING_ERRNO)
#undef SQRT_TYPE
DEF_LIB_BUILTIN (BUILT_IN_TAN, "tan", BT_FN_DOUBLE_DOUBLE, ATTR_MATHFN_FPROUNDING)
```

- Custom builtins in gcc/config/<arch>
  - e.g. gcc/config/riscv/riscv-builtins.cc

Why not Inline Assembler?

- GCC allows assembly code to be specified inline

```
__asm__ ("cv.elw %0,%1" : "=r"(res) : "m"(sema));
```

  - a “black box”, with limited dataflow information
  - optimization limited to selection of arguments
- Builtin functions offer some advantages
  - full dataflow information for optimization
  - patterns can be recognized and used elsewhere by GCC
  - generally available to be optimized.

CV32E40P ISA Extensions

- Standard RV32IMAFC core with 8 ISA extensions
  - Post Incrementing Load/Store
  - Hardware Loops
  - Immediate Branching
  - General ALU Operations
  - Multiply-Accumulate
  - Event Load
  - SIMD
  - Bit Manipulation

} Version 2 only

The Event Load Word Builtin

- The generated code (with -O0):

```
lui     a5,%hi(sema)
addi    a5,a5,%lo(sema)
cv.elw  a5,0(a5)
mv      a0,a5
```

- The generated code (with -O2):

```
lui     a5,%hi(sema)
cv.elw  a0,%lo(sema)(a5)
```

The Event Load Word Instruction

The event load instruction **cv.elw** is only supported if the `PULP_CLUSTER` parameter is set to 1. The event load performs a load word and can cause the CV32E40P to enter a sleep state as explained in PULP Cluster Extension.

Load Operations

Mnemonic	Description
Event Load	
cv.elw rD, Imm(rs1)	rD = Mem32(Sext(Imm)+rs1)

Encoding

31 : 20	19 :15	14 : 12	11 :07	06 : 00	
imm[11:0]	**rs1**	**funct3**	**rd**	**opcode**	**Mnemonic**
offset	base	011	dest	000 1011	**cv.elw rD, Imm(rs1)**

Naming Builtins

- General convention
  - __builtin_<name>
- Architecture specific standard naming
  - __builtin_<arch>_<name>
- Vendor specific RISC-V naming
  - __builtin_riscv_<vendor>_<name>
- CORE-V builtin naming
  - __builtin_riscv_cv_<isaext>_<name>
- cv.elw builtin name:
  - __builtin_riscv_cv_elw_elw

The Event Load Word Builtin

- Specification:

```
uint32_t __builtin_riscv_cv_elw_elw (void *)
```

- Example:

```
uint32_t sema;

uint32_t foo (void) {
  return __builtin_riscv_cv_elw_elw (&sema);
}
```

How to Add Extensions to GCC

- gcc/common/config/riscv/riscv-common.cc:

```
/* All standard extensions defined in all supported ISA spec. */
static const struct riscv_ext_version riscv_ext_version_table[] =
{
  /* name, ISA spec, major version, minor_version. */
  {"xcv", ISA_SPEC_CLASS_NONE, 1, 0},
  {"xcvelw", ISA_SPEC_CLASS_NONE, 1, 0},
  {"xcvsimd", ISA_SPEC_CLASS_NONE, 1, 0},
};
```

How to Add Extensions to GCC

- gcc/common/config/riscv/riscv-common.cc:

```
/* Implied ISA info, must end with NULL sentinel. */
static const riscv_implied_info_t riscv_implied_info[] =
{
  ...
 {"xcvelw", "xcv"},
 {"xcvsimd", "xcv"},
 {NULL, NULL}
};
```

How to Add Extensions to GCC

- gcc/config/riscv/riscv-opt.h:

```
#define MASK_XCOREV (1 << 0)
#define MASK_XCOREVELW (1 << 1)
#define MASK_XCOREVSIMD (1 << 2)

#define TARGET_XCOREV ((riscv_xcorev_flags & MASK_XCOREV) != 0)
#define TARGET_XCOREVELW ((riscv_xcorev_flags & MASK_XCOREVELW) != 0)
#define TARGET_XCOREVSIMD ((riscv_xcorev_flags & MASK_XCOREVSIMD) != 0)
```

- gcc/config/riscv/riscv.opt:

```
TargetVariable
int riscv_xcorev_flags
```

How to Add Extensions to GCC

- gcc/common/config/riscv/riscv-common.cc:

```
/* Mapping table between extension to internal flag. */
static const riscv_ext_flag_table_t riscv_ext_flag_table[] =
{
  {"xcv", &gcc_options::x_riscv_xcorev_flags, MASK_XCOREV},
  {"xcvelw", &gcc_options::x_riscv_xcorev_flags, MASK_XCOREVELW},
  {"xcvsimd", &gcc_options::x_riscv_xcorev_flags, MASK_XCOREVSIMD},
};
```

How to Define a RISC-V Builtin

- gcc/config/riscv/riscv-builtin.cc:

```
#define RISCV_BUILTIN(INSN, NAME, BUILTIN_TYPE, FUNCTION_TYPE, AVAIL) \
{ CODE_FOR_riscv_ ## INSN, "__builtin_riscv_" NAME, \
BUILTIN_TYPE, FUNCTION_TYPE, riscv_builtin_avail_ ## AVAIL }
```

INSN: the name of the associated instruction pattern in the machine description file

NAME: the name of the builtin function itself

BUILTIN_TYPE:

RISCV_BUILTIN_DIRECT or RISCV_BUILTIN_DIRECT_NO_TARGET

FUNCTION_TYPE: the return type and argument types

AVAIL: the name of the availability predicate

How to Define a RISC-V Builtin

- gcc/config/riscv/riscv-builtin.cc:

```
#define RISCV_BUILTIN(INSN, NAME, BUILTIN_TYPE, FUNCTION_TYPE, AVAIL) \
{ CODE_FOR_riscv_ ## INSN, "__builtin_riscv_" NAME, \
BUILTIN_TYPE, FUNCTION_TYPE, riscv_builtin_avail_ ## AVAIL }
```

- gcc/config/riscv/corev.def:

```
RISCV_BUILTIN (cv_elw_si, "cv_elw_elw", RISCV_BUILTIN_DIRECT,
        RISCV_USI_FTYPE_VOID_PTR, cvelw),
```

Instruction Patterns

- INSN: the name of the associated instruction pattern in the machine description file
- define_insn specifies patterns, against which insns are matched

```
(define_insn
    "name"
    RTL template
    condition
    output template
/   nsn attributes)
```

- RISC-V specific patterns in gcc/config/riscv/riscv.md:

Instruction Patterns

- Event Load Word builtin pattern:

```
(define_insn "riscv_cv_elw_si"
    [(set (match_operand:SI 0 "register_operand" "=r")
                  (unspec_volatile [(mem:SI (match_operand:SI 1
                                  "address_operand" "p"))]
                                  UNSPECV_CV_ELW))]

"TARGET_XCOREVELW && !TARGET_64BIT"
"cv.elw\t%0,%a1"

[(set_attr "type" "load")
 (set_attr "mode" "SI")])
```

Instruction Patterns: RTL Template

```
[(set (match_operand:SI 0 "register_operand" "=r")
      (unspec_volatile [(mem:SI (match_operand:SI 1
                                  "address_operand" "p"))]
                       UNSPECV_CV_ELW))]
```

- set has the general form (set *lval x*)
- store a value at an address
  - *lval* is the destination to write
  - *x* is the value to be written

Instruction Patterns: RTL Template

```
[(set (match_operand:SI 0 "register_operand" "=r")
     (unspec_volatile [(mem:SI (match_operand:SI 1
                                 "address_operand" "p"))]
                      UNSPECV_CV_ELW))]
```

- match_operand has the general form
  - (match_operand:*m n predicate constraint)*
- Pattern for an operand
  - *m* - the machine mode (type)
  - *n* - the index of this operand (used later)
  - *predicate* - true/false test if putative operand is OK
  - *constraint* - scheduler will try to fit to this

Instruction Patterns: RTL Template

```
[(set (match_operand:SI 0 "register_operand" "=r")
      (unspec_volatile [(mem:SI (match_operand:SI 1
                                 "address_operand" "p"))]
                       UNSPECV_CV_ELW))]
```

- match_operand yields an address
- mem:SI specifies the size of the object referenced
  - mode SI, i.e. 32-bits

Instruction Patterns: RTL Template

```
[(set (match_operand:SI 0 "register_operand" "=r")
     (unspec_volatile [(mem:SI (match_operand:SI 1
                                 "address_operand" "p"))]
                      UNSPECV_CV_ELW))]
```

- unspec_volatile indicates a side effect
  - volatile operation or operations that trap
  - UNSPEC_CV_ELW is an index to specify which

Instruction Patterns

- Event Load Word builtin pattern:

```
(define_insn "riscv_cv_elw_si"
    [(set (match_operand:SI 0 "register_operand" "=r")
                  (unspec_volatile [(mem:SI (match_operand:SI 1
                                  "address_operand" "p"))]
                                  UNSPECV_CV_ELW))]

"TARGET_XCOREVELW && !TARGET_64BIT"
"cv.elw\t%0,%a1"

[(set_attr "type" "load")
 (set_attr "mode" "SI")])
```

Instruction Patterns: Condition

"TARGET_XCOREVELW && !TARGET_64BIT"

- Top level predicate on whether this pattern can be used
- C expression
- Used for command line flags controlling patterns

Instruction Patterns

- Event Load Word builtin pattern:

```
(define_insn "riscv_cv_elw_si"
    [(set (match_operand:SI 0 "register_operand" "=r")
                  (unspec_volatile [(mem:SI (match_operand:SI 1
                                  "address_operand" "p"))]
                                  UNSPECV_CV_ELW))]

"TARGET_XCOREVELW && !TARGET_64BIT"
"cv.elw\t%0,%a1"

[(set_attr "type" "load")
 (set_attr "mode" "SI")])
```

Instruction Patterns: Output Template

```
"cv.elw\t%0,%a1"
```

- A string or a fragment of C code returning a string
- Refer to matched operands using %*index*
  - *index* is that used in match_operand
- Add character for fine control
  - %a*index* to substitute as a memory reference

Instruction Patterns

- Event Load Word builtin pattern:

```
(define_insn "riscv_cv_elw_si"
    [(set (match_operand:SI 0 "register_operand" "=r")
                  (unspec_volatile [(mem:SI (match_operand:SI 1
                                  "address_operand" "p"))]
                                  UNSPECV_CV_ELW))]

"TARGET_XCOREVELW && !TARGET_64BIT"
"cv.elw\t%0,%a1"

[(set_attr "type" "load")
 (set_attr "mode" "SI")])
```

Instruction Patterns: Attribute

```
[(set_attr "type" "load")
 (set_attr "mode" "SI")])
```

- Attributes that can be associated with a pattern
- No direct effect on pattern matching
- Can be queried in code for optimization passes etc.

Instruction Patterns

- Event Load Word builtin pattern:

```
(define_insn "riscv_cv_elw_si"
    [(set (match_operand:SI 0 "register_operand" "=r")
                  (unspec_volatile [(mem:SI (match_operand:SI 1
                                  "address_operand" "p"))]
                                  UNSPECV_CV_ELW))]

"TARGET_XCOREVELW && !TARGET_64BIT"
"cv.elw\t%0,%a1"

[(set_attr "type" "load")
 (set_attr "mode" "SI")])
```

How to Define a RISC-V Builtin

- gcc/config/riscv/riscv-builtin.cc:

```
#define RISCV_BUILTIN(INSN, NAME, BUILTIN_TYPE, FUNCTION_TYPE, AVAIL) \
{ CODE_FOR_riscv_ ## INSN, "__builtin_riscv_" NAME, \
BUILTIN_TYPE, FUNCTION_TYPE, riscv_builtin_avail_ ## AVAIL }
```

INSN: the name of the associated instruction pattern in the machine description file

NAME: the name of the builtin function itself

BUILTIN_TYPE:

RISCV_BUILTIN_DIRECT or RISCV_BUILTIN_DIRECT_NO_TARGET

FUNCTION_TYPE: the return type and argument types

AVAIL: the name of the availability predicate

RISC-V Builtin Types

- Type available:
  - RISCV_BUILTIN_DIRECT
  - RISCV_BUILTIN_DIRECT_NO_TARGET

- gcc/config/riscv/riscv-builtin.cc:

```
/* Specifies how a built-in function should be converted into rtl. */
enum riscv_builtin_type {
        /* The function corresponds directly to an .md pattern. */
        RISCV_BUILTIN_DIRECT,
        /* Likewise, but with return type VOID. */
        RISCV_BUILTIN_DIRECT_NO_TARGET
};
```

RISC_V Function Types

- The return and argument types of the builtin
- gcc/config/riscv/riscv-builtins.cc:

```
/* Macros to create an enumeration identifier for a function prototype. */
#define RISCV_FTYPE_NAME0(A) RISCV_##A##_FTYPE
#define RISCV_FTYPE_NAME1(A, B) RISCV_##A##_FTYPE_##B

/* RISCV_FTYPE_ATYPESN takes N RISCV_FTYPES-like type codes and lists
their associated RISCV_ATYPEs. */
#define RISCV_FTYPE_ATYPES0(A) \
   RISCV_ATYPE_##A
#define RISCV_FTYPE_ATYPES1(A, B) \
         RISCV_ATYPE_##A, RISCV_ATYPE_##B
```

RISC-V Function Types

- gcc/config/riscv/riscv-ftypes.def:

```
DEF_RISCV_FTYPE (0, (USI))
DEF_RISCV_FTYPE (1, (VOID, USI))
DEF_RISCV_FTYPE (1, (USI, VOID_PTR)) //RISCV_USI_FTYPE_VOID_PTR
```

- gcc/config/riscv/riscv-builtins.cc:

```
#define RISCV_ATYPE_VOID void_type_node
#define RISCV_ATYPE_USI unsigned_intSI_type_node
#define RISCV_ATYPE_VOID_PTR ptr_type_node
```

RISC-V Availability Predicate

- gcc/config/riscv/riscv-builtin.cc:

```
/* Declare an availability predicate for built-in functions. */
#define AVAIL(NAME, COND)                       \
static unsigned int                             \
riscv_builtin_avail_##NAME (void)               \
{                                               \
return (COND);                                  \
}

AVAIL (hard_float, TARGET_HARD_FLOAT)
//COREV AVAIL
AVAIL (cvelw, TARGET_XCOREVELW && !TARGET_64BIT)
AVAIL (cvsimd, TARGET_XCOREVSIMD && !TARGET_64BIT)
```

How to test a RISC-V builtin in GCC

- gcc/testsuite/gcc.target/riscv/cv-elw-compile-1.c:

```
/* { dg-do compile } */
/* { dg-options "-march=rv32i_xcvelw -mabi=ilp32" } */


int foo1(void* b)
{
return __builtin_riscv_cv_elw_elw(b+8);
}

/* { dg-final { scan-assembler-times "cv\\.elw" 1 } } */
```

- Run test with make check

How to test a RISC-V builtin in GCC

- <buildir>/gcc/testsuite/gcc/gcc.{log,sum}:

```
PASS: gcc.target/riscv/cv-elw-compile-1.c -O0 scan-assembler-times cv\\.elw 1
PASS: gcc.target/riscv/cv-elw-compile-1.c -O0 (test for excess errors)
PASS: gcc.target/riscv/cv-elw-compile-1.c -O1 scan-assembler-times cv\\.elw 1
PASS: gcc.target/riscv/cv-elw-compile-1.c -O1 (test for excess errors)
PASS: gcc.target/riscv/cv-elw-compile-1.c -O2 scan-assembler-times cv\\.elw 1
PASS: gcc.target/riscv/cv-elw-compile-1.c -O2 (test for excess errors)
PASS: gcc.target/riscv/cv-elw-compile-1.c -O2 -flto -fno-use-linker-plugin -flto-partition=none scan-assembler-times cv\\.elw 1
PASS: gcc.target/riscv/cv-elw-compile-1.c -O2 -flto -fno-use-linker-plugin -flto-partition=none (test for excess errors)
PASS: gcc.target/riscv/cv-elw-compile-1.c -O2 -flto -fuse-linker-plugin -fno-fat-lto-objects scan-assembler-times cv\\.elw 1
PASS: gcc.target/riscv/cv-elw-compile-1.c -O2 -flto -fuse-linker-plugin -fno-fat-lto-objects (test for excess errors)
PASS: gcc.target/riscv/cv-elw-compile-1.c -O3 -g scan-assembler-times cv\\.elw 1
PASS: gcc.target/riscv/cv-elw-compile-1.c -O3 -g (test for excess errors)
PASS: gcc.target/riscv/cv-elw-compile-1.c -Os scan-assembler-times cv\\.elw 1
PASS: gcc.target/riscv/cv-elw-compile-1.c -Os (test for excess errors)
PASS: gcc.target/riscv/cv-elw-compile-1.c -Og -g scan-assembler-times cv\\.elw 1
PASS: gcc.target/riscv/cv-elw-compile-1.c -Og -g (test for excess errors)
PASS: gcc.target/riscv/cv-elw-compile-1.c -Oz scan-assembler-times cv\\.elw 1
PASS: gcc.target/riscv/cv-elw-compile-1.c  -Oz  (test for excess errors)
```

Reference Material

- CORE-V source code
  - github.com/openhwgroup/corev-binutils-gdb
  - github.com/openhwgroup/corev-gcc
- The Open Hardware Group
  - www.openhwgroup.org/
- GCC Internals Manual
  - gcc.gnu.org/onlinedocs/gccint/index.html

Thank you

nandni.jamnadas@embecosm.com
www.embecosm.com

github.com/NandniJamnadas

Nandni Jamnadas