



How to Add a Builtin Function to the GCC backend

Jeremy Bennett

Copyright © 2025 Embecosm. Freely available under a
Creative Commons Attribution-ShareAlike license.



Three Numbers

26.9 million

10,174

49

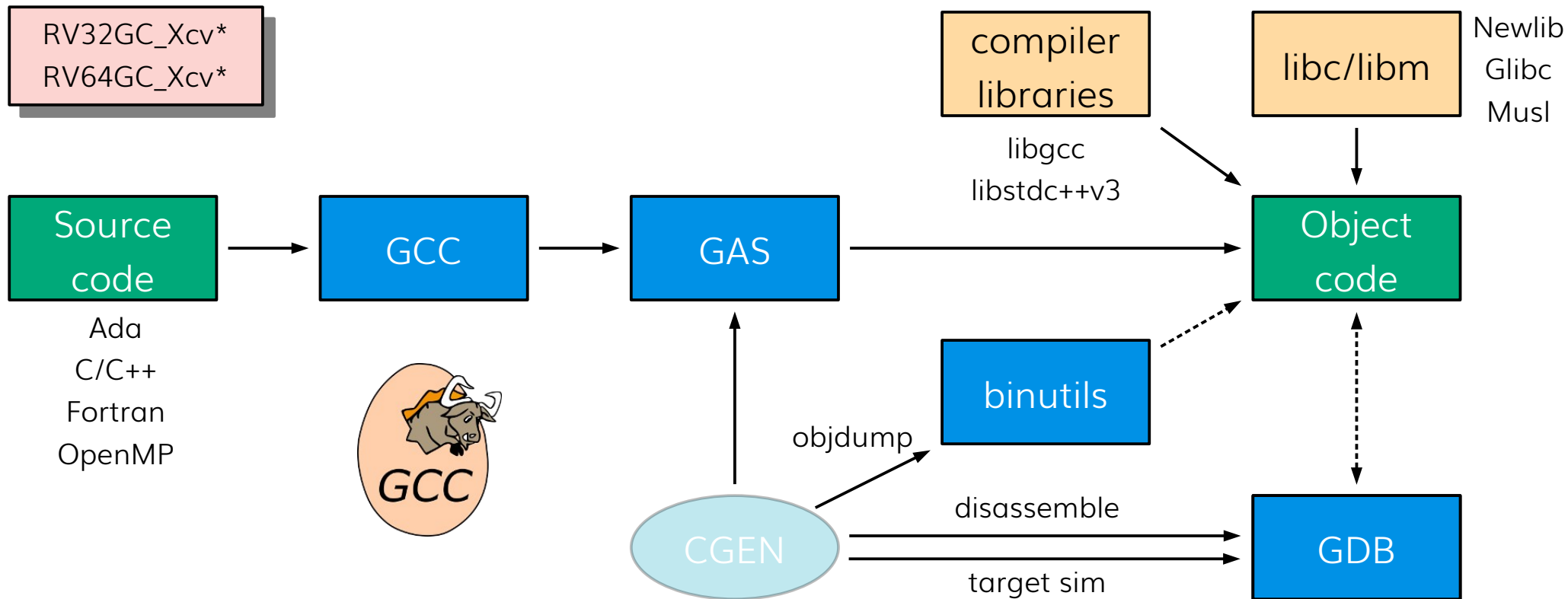


Overview

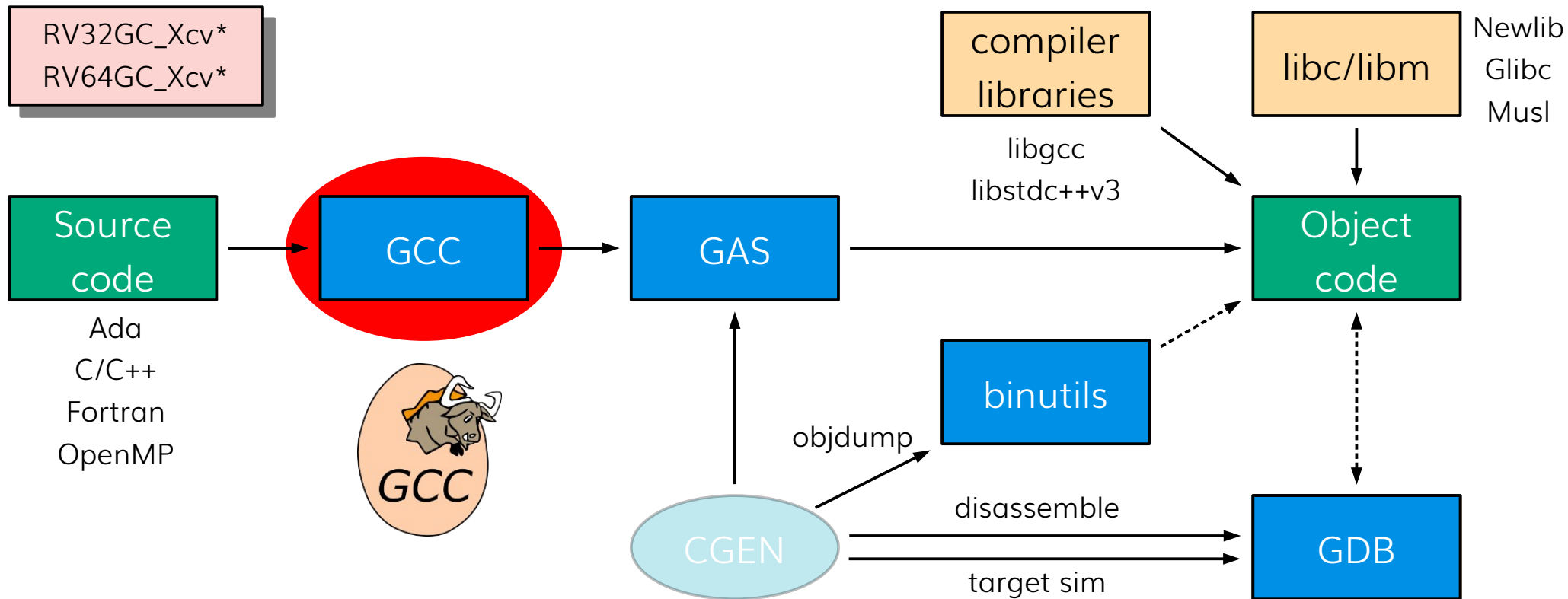
Copyright © 2025 Embecoscsm. Freely available under a
Creative Commons Attribution-ShareAlike license.



CORE-V GNU Tool Chain Components



CORE-V GNU Tool Chain Components



CORE-V ISA Extensions

- **Xcvmem**: Post-incrementing load/store (25)
- **Xcvhwl**: Hardware loops (6)
- **Xcval**: General ALU operations (31)
- **Xcvbi**: Immediate branching operations (2)
- **Xcvmac**: Multiply-accumulate (22)
- **Xcvel**: Event Load (1)
- **Xcvbitmanip**: PULP Bit manipulation (16)
- **Xcvsimd**: PULP SIMD (220)



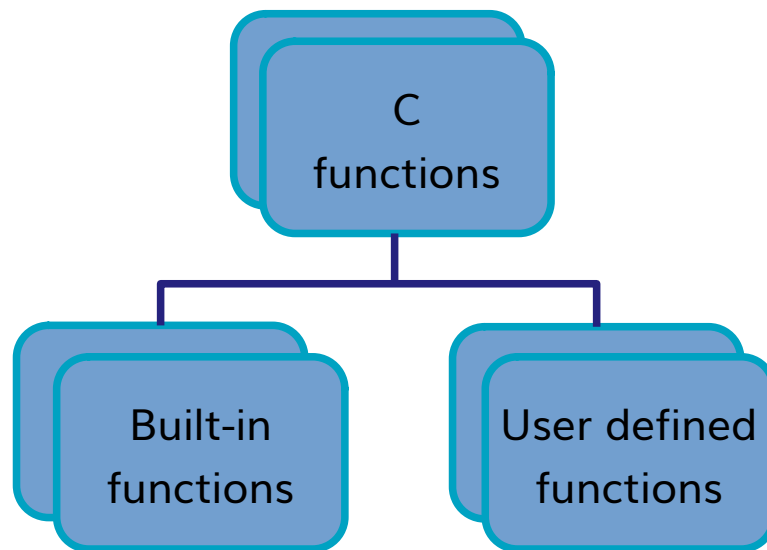
What is a Builtin Function?

Copyright © 2025 Embecosm. Freely available under a
Creative Commons Attribution-ShareAlike license.



What is a “Builtin”?

- 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 CORE-V, used to expose ISA extension functionalities



What is a “Builtin”

```
/* 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_FROUNDING)  
DEF_LIB_BUILTIN (BUILT_IN SINH, "sinh", BT_FN_DOUBLE_DOUBLE, ATTR_MATHFN_FROUNDING_ERRNO)  
DEF_C99_C90RES_BUILTIN (BUILT_IN_SINHF, "sinhf", BT_FN_FLOAT_FLOAT, ATTR_MATHFN_FROUNDING_ERRNO)  
DEF_C99_C90RES_BUILTIN (BUILT_IN_SINHL, "sinhl", BT_FN_LONGDOUBLE_LONGDOUBLE, ATTR_MATHFN_FROUNDING_ERRNO)  
DEF_C99_C90RES_BUILTIN (BUILT_IN_SINL, "sinl", BT_FN_LONGDOUBLE_LONGDOUBLE, ATTR_MATHFN_FROUNDING)  
DEF_LIB_BUILTIN (BUILT_IN_SQRT, "sqrt", BT_FN_DOUBLE_DOUBLE, ATTR_MATHFN_FROUNDING_ERRNO)  
DEF_C99_C90RES_BUILTIN (BUILT_IN_SQRTF, "sqrtf", BT_FN_FLOAT_FLOAT, ATTR_MATHFN_FROUNDING_ERRNO)  
DEF_C99_C90RES_BUILTIN (BUILT_IN_SQRTL, "sqrtl", BT_FN_LONGDOUBLE_LONGDOUBLE, ATTR_MATHFN_FROUNDING_ERRNO)  
#define Sqrt_Type(F) BT_FN_##F##_##F  
DEF_EXT_LIB_FLOATN_NX_BUILTINS (BUILT_IN_SQRT, "sqrt", Sqrt_Type, ATTR_MATHFN_FROUNDING_ERRNO)  
#undef Sqrt_Type  
DEF_LIB_BUILTIN (BUILT_IN_TAN, "tan", BT_FN_DOUBLE_DOUBLE, ATTR_MATHFN_FROUNDING)
```

- 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



Implementing a Simple Builtin

Copyright © 2025 Embecosm. Freely available under a
Creative Commons Attribution-ShareAlike license.



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	
<code>cv.elw rD, Imm(rs1)</code>	$rD = \text{Mem32}(\text{Sext}(\text{Imm}) + rs1)$

Encoding

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

The Event Load Word Builtin

Source C code

```
extern int  sema;

int elw_func ()
{
    return
        __builtin_riscv_cv_elw_elw (&sema);
}
```

The Event Load Word Builtin

Source C code

```
extern int  sema;

int elw_func ()
{
    return
        __builtin_riscv_cv_elw_elw (&sema);
}
```

Generated code (with -O0)

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

The Event Load Word Builtin

Source C code

```
extern int  sema;

int elw_func ()
{
    return
        __builtin_riscv_cv_elw_elw (&sema);
}
```

Generated code (with **-O0**)

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

Generated code (with **-O2**)

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


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 Define a RISC-V Builtin

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

```
#define RISC_V_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 associated instruction pattern in the machine description file
- **NAME**: the name of the builtin function itself
- **BUILTIN_TYPE**:
 - `RISC_V_BUILTIN_DIRECT` or `RISC_V_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 RISC_V_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:`

```
RISC_V_BUILTIN (cv_elw_elw_si, "cv_elw_elw", RISC_V_BUILTIN_DIRECT,
  RISC_V_USI_FTYPE_VOID_PTR, cvelw)
```

Instruction Patterns

- **INSN**: the instruction pattern in the machine description
- **define_insn** specifies patterns, against which insns are matched

```
(define_insn  
  "name"  
  RTL template  
  condition  
  output template  
  I      insn attributes)
```

- CORE-V specific patterns in **gcc/config/riscv/corev.md**

Instruction Patterns

- Event Load Word builtin pattern:

```
(define_insn "riscv_cv_elw_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_XCVELW && !TARGET_64BIT"
"cv.elw\t%0,%a1"

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

Instruction Patterns

- Event Load Word builtin pattern:

Name

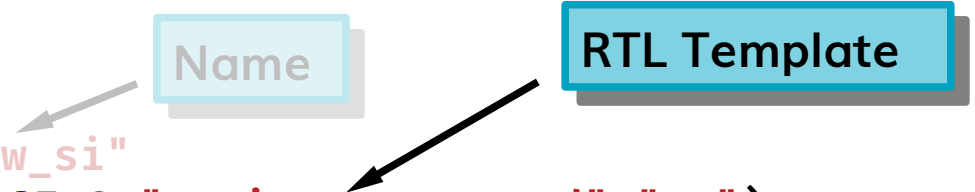
```
(define_insn "riscv_cv_elw_elw_si"  
  [(set (match_operand:SI 0 "register_operand" "=r")  
        (unspec_volatile [(mem:SI (match_operand:SI 1  
                                   "address_operand" "p"))]  
                          UNSPECV_CV_ELW))])
```

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

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

Instruction Patterns

- Event Load Word builtin pattern:

The diagram shows two light blue boxes at the top: 'Name' on the left and 'RTL Template' on the right. A grey arrow points from the 'Name' box to the instruction name 'riscv_cv_elw_elw_si' in the code. A black arrow points from the 'RTL Template' box to the entire instruction body, which is enclosed in square brackets.

```
(define_insn "riscv_cv_elw_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_XCVELW && !TARGET_64BIT"
"cv.elw\t%0,%a1"

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


Instruction Patterns

- Event Load Word builtin pattern:

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

Diagram labels and arrows:

- Name** points to `"riscv_cv_elw_elw_si"`
- RTL Template** points to the `(unspec_volatile ...)` block

Condition

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

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

Instruction Patterns

- Event Load Word builtin pattern:

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

Condition

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

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

Output Template

Instruction Patterns

- Event Load Word builtin pattern:

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

Condition

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

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

Output Template

Attributes

Instruction Patterns

- Event Load Word builtin pattern:

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

Condition

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

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

Name

RTL Template

Output Template

Attributes

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** operand pattern has the general form

(match_operand:m n predicate constraint)

- **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** dereferences this address
 - mode **SI**, indicates the size 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
- Used for simple builtins
 - no attempt to pattern match the internal structure

Instruction Patterns

- Event Load Word builtin pattern:

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

Condition

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

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

Name

RTL Template

Output Template

Attributes

Instruction Patterns: Condition

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

Condition

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

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

Output Template

Attributes

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

Condition

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

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

Output Template

Attributes

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.

How to Define a RISC-V Builtin

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

```
#define RISC_V_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 associated instruction pattern in the machine description file
- **NAME**: the name of the builtin function itself
- **BUILTIN_TYPE**:
 - `RISC_V_BUILTIN_DIRECT` or `RISC_V_BUILTIN_DIRECT_NO_TARGET`
- **FUNCTION_TYPE**: the return type and argument types
- **AVAIL**: the name of the availability predicate

RISC-V Builtin Types

- Types 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_ATYPE0(A) \  
    RISCV_ATYPE_##A
```

```
#define RISCV_FTYPE_ATYPE1(A, B) \  
    RISCV_ATYPE_##A, RISCV_ATYPE_##B
```

RISC-V Function Types

- `gcc/config/riscv/riscv-ftype.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_XCVELW && !TARGET_64BIT)
```

```
AVAIL (cvsimd, TARGET_XCVSIMD && !TARGET_64BIT)
```

The Event Load Word Builtin: Summary

`gcc/config/riscv/corev.def:`

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



More Complex Builtins

Copyright © 2025 Embecoscsm. Freely available under a
Creative Commons Attribution-ShareAlike license.



SIMD Scalar Add Byte Instruction

- Two assembly instructions
 - `cv.add.sc.b rd,rs1,rs2`
 - `cv.add.sci.b rd,rs1,imm6` $(-32 \leq \text{imm6} \leq 31)$
- One builtin
 - `uint32_t __builtin_riscv_cv_simd_add_sc_b (uint32_t i, int8_t j)`
- Builtin must generate code based on argument value
 - constant in range will generate `cv.add.sci.b`
 - everything else will move the second argument to a register

Define the Builtin

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

```
RISCV_BUILTIN (cv_simd_add_sc_b_si, "cv_simd_add_sc_b",  
               RISCV_BUILTIN_DIRECT, RISCV_USI_FTYPE_USI_QI, cvsimd)
```

- **INSN**: 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`
- **FUNCTION_TYPE**: the return type and argument types
- **AVAIL**: the name of the availability predicate

The Machine Pattern

```
(define_insn "riscv_cv_simd_add_sc_b_si"

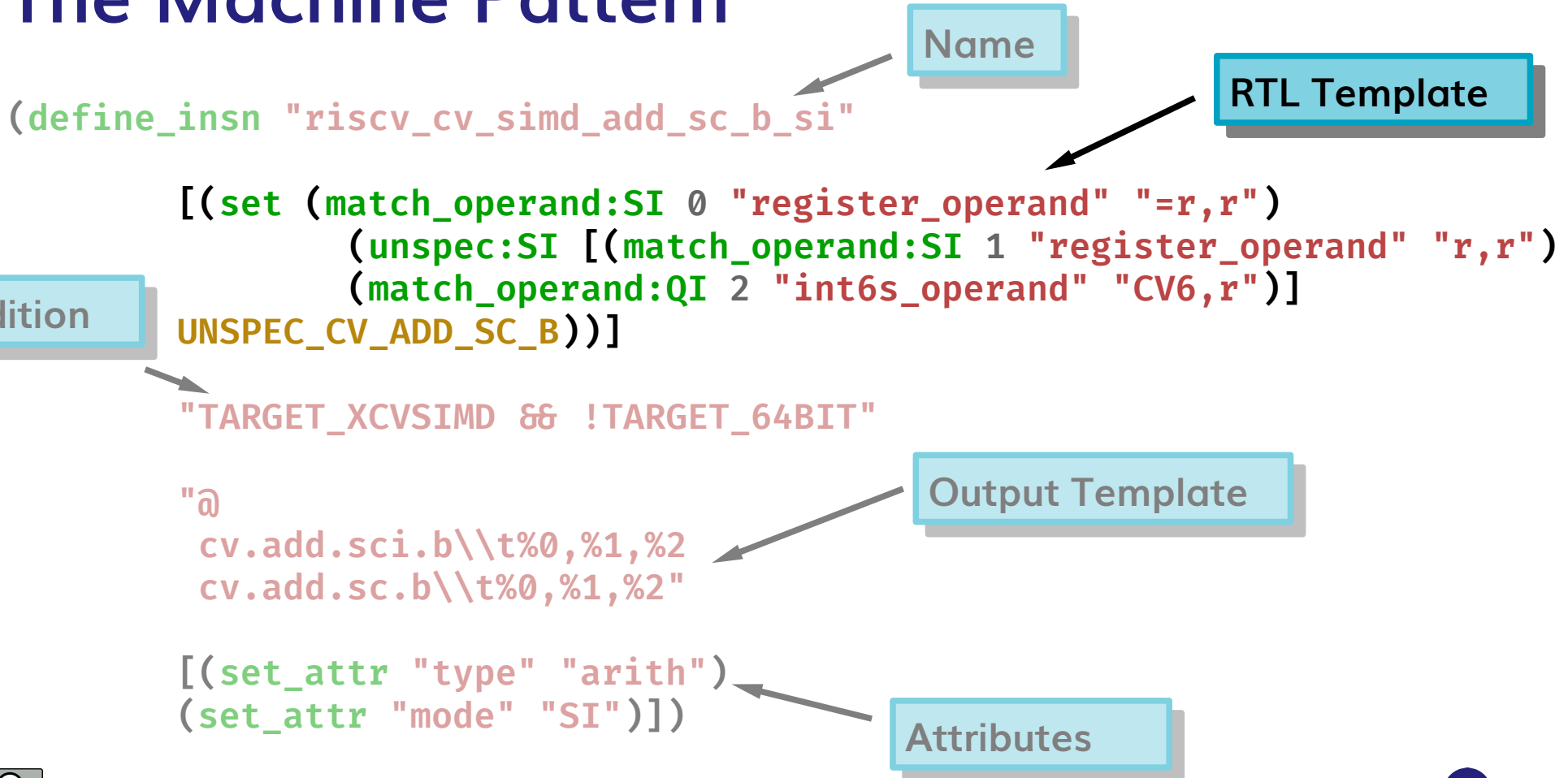
  [(set (match_operand:SI 0 "register_operand" "=r,r")
        (unspec:SI [(match_operand:SI 1 "register_operand" "r,r")
                     (match_operand:QI 2 "int6s_operand" "CV6,r")]
                     UNSPEC_CV_ADD_SC_B)))

  "TARGET_XCVSIMD && !TARGET_64BIT"

  "@
  cv.add.sci.b\\t%0,%1,%2
  cv.add.sc.b\\t%0,%1,%2"

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


The Machine Pattern



RTL Template

```
[(set (match_operand:SI 0 "register_operand" "=r,r")
      (unspec:SI [(match_operand:SI 1 "register_operand" "r,r")
                  (match_operand:QI 2 "int6s_operand" "CV6,r")]
                  UNSPEC_CV_ADD_SC_B)))]
```

- We now have pairs of constraints for each operand
 - first choice **=r, r** and **CV6**
 - second choice **=r, r** and **r**
 - we'll see the definition of **CV6** later

Custom Constraints

`gcc/config/riscv/constraints.md`

```
(define_constraint "CV6"  
  "A 6-bit signed immediate for SIMD."  
  (and (match_code "const_int")  
    (match_test "IN_RANGE (ival, -32, 31)"))))
```

- `match_code` to use existing constraints
- `match_test` to add a test
- combine with `and` and `or`

RTL Template

```
[(set (match_operand:SI 0 "register_operand" "=r,r")
      (unspec:SI [(match_operand:SI 1 "register_operand" "r,r")
                  (match_operand:QI 2 "int6s_operand" "CV6,r")]
        UNSPEC_CV_ADD_SC_B)))]
```

- We have a new predicate for the last operand
 - we'll see the definition of **int6s_operand** later

Custom Predicates

`gcc/config/riscv/predicates.md`

```
(define_predicate "const_int6s_operand"  
  (and (match_code "const_int")  
        (match_test "IN_RANGE (INTVAL (op), -32, 31)"))))
```

```
(define_predicate "int6s_operand"  
  (ior (match_operand 0 "const_int6s_operand")  
        (match_operand 0 "register_operand")))
```

The Machine Pattern

```
(define_insn "riscv_cv_simd_add_sc_b_si"
```

Name

RTL Template

```
  [(set (match_operand:SI 0 "register_operand" "=r,r")  
        (unspec:SI [(match_operand:SI 1 "register_operand" "r,r")  
                     (match_operand:QI 2 "int6s_operand" "CV6,r")]  
                     UNSPEC_CV_ADD_SC_B))]  
  UNSPEC_CV_ADD_SC_B))]
```

Condition

```
"TARGET_XCVSIMD && !TARGET_64BIT"
```

```
"@  
  cv.add.sci.b\\t%0,%1,%2  
  cv.add.sc.b\\t%0,%1,%2"
```

Output Template

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

Attributes

The Output Template

```
"@  
cv.add.sci.b\\t%0,%1,%2  
cv.add.sc.b\\t%0,%1,%2"
```

- introduced by @
- one pattern for each of the two sets of constraints
 - $\{=r, r, CV6\} \rightarrow cv.add.sci.b\\t%0,%1,%2$
 - $\{=r, r, r\} \rightarrow cv.add.sc.b\\t%0,%1,%2"$

Scalar Add Byte Example (-O2)

```
#include <stdint.h>
```

```
uint32_t val4;
```

```
uint32_t fv (uint8_t x)
```

```
{  
    return __builtin_riscv_cv_simd_add_sc_b (val4, x);    fv:  
}
```

```
lui    a5,%hi(val4)  
lw     a5,%lo(val4)(a5)  
cv.add.sc.b    a0,a5,a1  
ret
```

```
uint32_t fc (void)
```

```
{  
    return __builtin_riscv_cv_simd_add_sc_b (val4, 28);    fc:  
}
```

```
lui    a5,%hi(val4)  
lw     a0,%lo(val4)(a5)  
cv.add.sci.b    a0,a0,28  
ret
```


Reusing arguments: Example

```
(define_insn "riscv_cv_mac_mac"

  [(set (match_operand:SI 0 "register_operand" "=r")
        (fma:SI (match_operand:SI 1 "register_operand" "r")
                  (match_operand:SI 2 "register_operand" "r")
                  (match_operand:SI 3 "register_operand" "0"))))]

  "TARGET_XCVMAC && !TARGET_64BIT"

  "cv.mac\t%0,%1,%2"

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



What Next?

Copyright © 2025 Embecosm. Freely available under a
Creative Commons Attribution-ShareAlike license.



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

Getting Involved

- As a user
 - download the latest development tool chains
 - embecosm.com/resources/tool-chain-downloads
 - pre-built binaries, source code, scripts and test results
- As a developer
 - join the OpenHW Mattermost SW : GNU Tools channel
 - sign up to the OpenHW SW mailing list and attend the monthly meeting
 - submit your pull requests against the development branch
 - github.com/openhwgroup/corev-binutils-gdb
 - github.com/openhwgroup/corev-gcc

Acknowledgements

- The Embecosm CORE-V GCC team
 - Mary Bennett
 - Hélène Chelin
 - Pietra Ferreira
 - Nandni Jamnadas
 - Charlie Keaney
 - Jessica Mills



Thank You

jeremy.bennett@embecoscsm.com

embecoscsm.com