CHAPTER 1 CSL Instruction Set Architecture Generator

TABLE 1.1 Chapter Outline

1.1 CSL ISA Syntax and Command Summary
1.2 CSL ISA Commands

1.1 CSL ISA Syntax and Command Summary

1.1.1 ISA Classes

CSL ISA classes are used to specify Instruction Set Architectures (ISA). In CSL several types of classes are used when defining an ISA: CSL ISA element, CSL ISA field and CSL enum. These classes will form a tree-like structure that will generate the ISA. An ISA contains instruction formats and instructions which contain fields. We can associate an enumerated type with one or more fields of the instruction format. The ISA can be associated with a signal, port or register. The isa tree structure is used to generate decoders, selection signals, address signals, immediat constants etc which drive the processor datapath.

1.1.1.1 CSL ISA Field class

An instruction is formed with fields which hold various parts of the instruction (opcode, source/destination addresses, flags, etc..). In CSL these ranges are represented by a variation of the CSL Field class: CSL ISA Field.
The fields of csl_isa_elements of type format indicate how the bits of the instruction register are interpreted when different instruction are decoded. The interpretation defines the field type. Fields can have the following types:

TABLE 1.2

Field name	Description
opcode	opcode
subopcode	subopcode
rf_address	register file address
mem_address	memory address
branch_address	branch address
im	immediate constant
selector	selector
reserved	reserved by designer
unused	unused
constant	constant value

Field types can be used to make checks that the generated signals are connected to the right units (ex. register file address signals are connected to register file address ports and not to memory address ports etc);
Opcode - opcode used in the left hand field of an instruction, holds an enumerated type which contains opcodes.
Subopcode -subopcode associated with a sub field in an instruction format which is specified by an opcode.

Rf_address =register file address - used to hold a register file read or write address.
Mem_address = memory address - used to hold a memory read or write address.
Branch_address - used to hold branch address.
Im =immediate constant -holds a constant value that is loaded into the datapath and used in an operation or loaded into a register, register file, or memory location.
Mux_selector - a mux selector that selects a mux input.
Op_selector -selects an option in a unit such as an alu which supports multiple operations.
Reserved - reserved by designer constant constant value unused.
Unused - hold an unused field.
Constant - hold a constant value.

1.1.1.1.1 CSL ISA Field class declaration

CSL ISA Field are an exception from the usual CSL classes in the way they are declared: CSL ISA Fields have a dual object/class declaration mode. In most cases ISA Fields will be declared just like a bitrange object and a limited set of commands will be called upon (e.g. associating an enum). In this case, the declaration appears as the one for a CSL object with optional parameters as explained in each case below:

CSL ISA Field object declaration

```
csl_isa_field isa_field_object(width | bitrange[,enum]);
csl_isa_field isa_field_object(field_object_name);
csl_isa_field isa_field_object(lower, upper[,enum]);
```

In some instances, the user may require hierarchical ISA Fields. In this case, these are declared in the global scope just like any other CSL class. The ISA Field class declaration is shown in the below example:

```
csl_isa_field isa_field_name {
  (objects declarations/instantiations)+
  isa_field_name(){
    (isa_field methods calls)+
  }
};
```

Note for example above: **bold** text represents language reserved syntax, *italics* are user defined variables and blue text is a short BNF representation of CSL commands/declarations.

ISA Fields definition:
:

TABLE 1.3 ISA Field definition rules

CSL class	Uses CSL ISA Field
global scope	YES
CSL Unit	-

TABLE 1.3 ISA Field definition rules

CSL class	Uses CSL ISA Field
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL ISA Element	YES
CSL ISA Field	YES
CSL Field	-

In the ISA Field class' scope other fields can be instantiated or declared. ISA Fields are declared in another ISA Field using the object declaration syntax. Hierarchical fields can only be instantiated in another hierarchical field.

TABLE 1.4 Rules for instantiating objects in an ISA Field's scope

CSL class	Is instantiated in CSL ISA Fields scope
CSL Unit	-
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL ISA Element	-
CSL ISA Field	YES
CSL Field	-

1.1.1.1.2 CSL ISA Field usage and rules

ISA Fields are instantiated in ISA elements as discussed later. ISA Fields can also be instantiated in hierarchical ISA Fields. Table 1.5 shows the usage of ISA Fields related to other CSL classes:

TABLE 1.5 ISA Field usage rules

CSL class	Uses CSL ISA Field
CSL Unit	-
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL ISA Element	YES
CSL ISA Field	YES
CSL Field	-

Based on the field type, signals needed in the data path must be generated. The generation of the signals are made by using the command:

```
signal/port/register.generate_signals();
```

This command can be called on that object only if user has previously associated with that object an ISA. In this case the signal generated for a field can be referred in the design in the following way:

```
signal.instruction_format.field
```

EXAMPLE :

In this example there is a signal named *ir* that is associated with an ISA .

```
ir.base_format.opcode
```

ISA signal corresponding to the opcode field of the instruction. *ir* is the signal to which the isa has been associated.

```
ir.aluk_format.kfield
```

Signal corresponding to the field containing the immediate constant for an arithmetic instruction.

Some fields of the instruction register are decoded. The CSL compiler will generate the correspond-

ing decoders for fields/enumerated types . Fields that will be decoded may have associated an enum that will be used to generate the names of the decoder outputs.

The generation of decoders is specified using the command *gen_decoder()*.
The command bellow is used to generate all the decoders specified in the ISA (all fields which have associated enums).

```
signal/port/register.generate_decoder();
```

This command can be called on that object only if user has previously associated with that object an ISA.
If user doesn't need to generate all decoders corresponding to the fields that have an associated enum, the following command can be used:

```
signal/port/register.instruction_format.field.generate_decoder();
```

This command can be called on any object only if user has previously associated with that object an ISA.

For each field that has associated an enumerated type a decoder can be generated. In this case the output signals of the decoder will be referred in the design in the following way:

```
signal/port/register.instruction_format.field[enum_item];
```

EXAMPLE :

```
ir.base_format.opcode[ADD]
```

ISA signal that will be active when the ADD instruction is selected by the decoder.

```
ir.jcrbase_format.cfield[NC]
```

ISA signal that will be active when the conditional jump if not carry instruction is selected by the decoder.

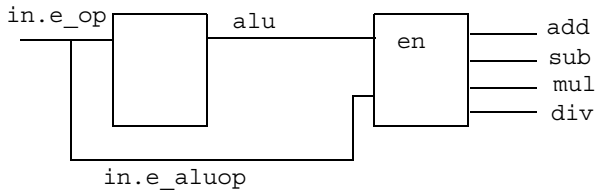
```
ir.base_format.opcode[STORE]
```

ISA signal that will be active when the store to memory instruction is active.

Sometimes, an ISA can define a second field (subopcode) to create a subfamily of instructions. In this case the generated decoder for the subopcode field must be qualified by the decode of the opcode for the instruction format that contains the subopcode field. CSL ISA specification automatically generates the qualification signal for the generated subopcode decoders.

EXAMPLE :

FIGURE 1.1



CSL CODE

```
csl_enum e_op{
  ALU,
  MOV,
  BR
};
csl_enum e_aluop{
  XOR,
  OR,
  AND
};
csl_isa_field f_op(2, e_op);
f_op.set_type(opcode);

csl_isa_field f_subop(2, e_aluop);
f_subop.set_type(opcode);

csl_isa_element isa {
  isa(){
    set_type(root_format);
    set_width(32);
  }
};
csl_isa_element base_format: isa {
  f_op opcode;
  base_format() {
    set_type(inst_format);
    set_position(opcode, 30);
  }
};
```

```
csl_isa_element alu_format : base_format {
  f_subop subop;
  alu_format() {
    set_type(inst_format);
    opcode.set_enum_item(ALU);
    set_position(subop, 16);
  }
};
.....................................
csl_signal ir(32);
ir.set_isa(isa);

ir.base_format.opcode.gen_decoder();
ir.alu_format.f_subop.gen_decoder();
```

The opcode qualification of subopcodes will be generated automatically.The Verilog RTL code for the above CSL code looks like the following:

VERILOG CODE

```
`define ALU 2'h0
`define MOV 2'h1
`define BR  2'h2

`define XOR 2'h0
`define OR  2'h1
`define AND 2'h2

`define F_OPCODE_UPPER_INDEX 31
`define F_OPCODE_LOWER_INDEX 30
`define F_SUBOP_UPPER_INDEX  17
`define F_SUBOP_LOWER_INDEX  16
.............................................................

wire [F_OPCODE_UPPER_INDEX - F_OPCODE_LOWER_INDEX : 0] opcode =
    s_ir[[F_OPCODE_UPPER_INDEX : F_OPCODE_LOWER_INDEX];

wire [F_SUBOP_UPPER_INDEX - F_SUBOP_LOWER_INDEX : 0] subop =
    s_ir[F_SUBOP_UPPER_INDEX : F_SUBOP_LOWER_INDEX];
```

```
wire [3:0] dec_opcode = 1'b1 << opcode;

wire dec_opcode_alu   = dec_opcode[ADD];
wire dec_opcode_mov = dec_opcode[MOV];
wire dec_opcode_br     = dec_opcode[BR];

wire [3:0] dec_subop =  dec_opcode[ALU]  & (1'b1 << subop); // qualify
the sub opcode decode

wire dec_subop_xor  = dec_subop[XOR];
wire dec_subop_or    = dec_subop[OR];
wire dec_subop_and  = dec_subop[AND];
```

1.1.1.1.3 ISA VLIW

VLIW instructions encode multiple operations that can be executed by the execution units of the device. The execution units of the device can be the same (equivalent to running multiple identical RISC-like execution units in parallel) or can be different.

If the execution units are the same, the VLIW instruction is specified by concatenating multiple copies of the same instruction format matching the number of execution units.

Specification of the VLIW format is done by defining a csl_isa_element of type vliw_format.

EXAMPLE :

Define a VLIW format for 4 identical execution units

// Specify the standard instruction format

```
csl_isa_element isa{
  isa(){
    set_type(root_format);
    set_width(32);
    }
};

.....................
```

//specification of the isa tree

```
.....................
```

// Specify the VLIW instruction format as the concatenation of
// four copies of the standard instruction format

```
csl_isa_element vliw_isa{
```

```
  isa isa0, isa1, isa2, isa3; // four instances of the same
                              // standard format
  vliw_isa()
    set_type(vliw_format);
    set_width(128);      //can be inferred from the format instances
    set_position(isa0,0);  // not necessary needed as position
                           // can be inferred from instantiation
    set_position(isa1,32);
    set_position(isa2,64);
    set_position(isa3,96);
    }
};
```

In some VLIW implementations the set of instructions available in a normal superscalar processor are partitioned, so different slots of the VLIW instructions are reserved only for a specific set of instructions (ex. fix point arithmetic, floating point arithmetic). For this case, in CSL we can define multiple root formats (that implement only specific set of instructions) and instantiate them in the csl_isa_element of type vliw_format.

EXAMPLE :

```
csl_isa_element isa_base{ // standard instruction set, no arithmetic
ops
   isa_base(){
     set_width(32);
     set_type(root_format);
     }
};
...
```

//specify the isa_base tree

```
...
csl_isa_element isa_fix_arith{ // restricted to fixed point arithmetic
   isa_fix_arith(){
     set_width(32);
     set_type(root_format);
     }
};
...
```

//specify the isa_fix_arith tree

```
...
csl_isa_element isa_float_arith{ // restricted to floating point
arithmetic
```

```
   isa_float_arith(){
     set_width(32);
     set_type(root_format);
     }
};
...
```

//specify the isa_float_arith tree

```
...

csl_isa_element vliw_isa{
  isa_base isa0, isa1; //two standard instruction slots
  isa_fix_arith isa2;  //fixed point instruction slot
  isa_float_arith isa3; //floating point instruction slot
  vliw_isa()
     set_type(vliw_format);
     set_width(128); //if not inferred from the format instances
     set_position(isa0,0); // needed if different from the
                           // instantiation order
     set_position(isa1,32);
     set_position(isa2,64);
     set_position(isa3,96);
     }
};
```

1.1.1.1.4 CSL ISA Field class commands

The following commands can be called on both ISA Field class (inside the constructor) and on ISA Fields declared using an object syntax.

CSL ISA Field class commands
[isa_field_obj_name.]set_type(isa_field_type);
[isa_field_obj_name.]set_name(string);
isa_field_obj_name.set_offset(num_expr);
isa_field_obj_name.set_enum(enum_name);
isa_field_instance_obj_name.set_enum_item(enum_item_name);
isa_field_obj_name.set_value(num_expr);
set_position(isa_field, numeric_expression);
set_next(csl_isa_field left_field, csl_isa_field right_field);
set_previous(csl_isa_field left_field, csl_isa_field right_field);

CSL ISA Field class commands

```
set_width (numeric_expression);
isa_field_obj_name.set_mnemonic(string);
[signal/register.]gen_decoder(csl_unit unit_object_name);
isa_field_obj_name.add_allowed_range(lower_num_expr, upper_num_expr);
```

1.1.2 Usage tables

CSL ISA element
can be defined and inherited
- can be defined in

TABLE 1.6 CSL ISA element definition in other CSL classes

CSL class	Can be defined in
global scope	YES
CSL Unit	-
CSL Signal Group	-
CSL Interface	-
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL ISA Element	-
CSL ISA Field	-
CSL Field	-

- can be inherited from

TABLE 1.7 CSL ISA element inheritance rules

CSL class	Can be inherited from
global scope	-
CSL Unit	-

TABLE 1.7 CSL ISA element inheritance rules

CSL class	Can be inherited from
CSL Signal Group	-
CSL Interface	-
CSL Testbench	-
CSL Vector	-
CSL State Data	-
CSL Register	-
CSL Register File	-
CSL Fifo	-
CSL Memory Map	-
CSL Memory Map Page	-
CSL ISA Element	YES
CSL ISA Field	-
CSL Field	-

CSL_ISA_FIELD METHODS

CSL_ISA_ELEMENT

```
csl_isa_element isa_obj_name [: isa_obj_name0];
```

CSL_ISA_ELEMENT METHODS

```
set_type (instr_format | instr | root_format | vliw_format );
set_width (numeric_expression);
set_position(isa_field, numeric_expression);
set_next(csl_isa_field left_field, csl_isa_field right_field);
set_previous(csl_isa_field left_field, csl_isa_field right_field);
```

Associate ISA to a register/signal

```
set_isa(isa_root_format_name);
[register/signal.]gen_signals();
```

1.2 CSL ISA Commands

CSL ISA INSTRUCTION FORMAT

csl_isa_field *isa_field_object_name;*

DESCRIPTION :

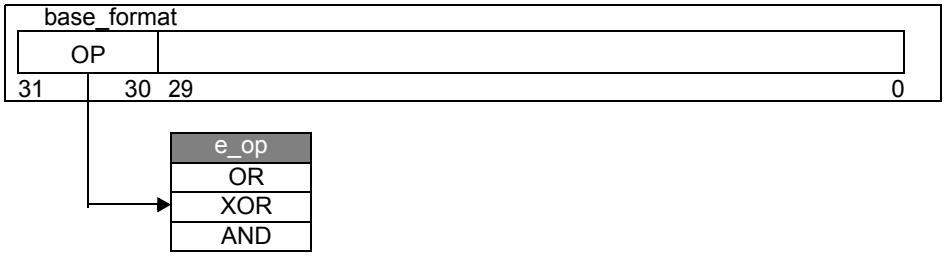
Creates a ISA field *isa_field_object.*

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

In this example a field object named ***f_op*** is declared and associated with the enum e_op.

FIGURE 1.2



CSL CODE

```
csl_enum e_op{
 OR,
 XOR,
 AND
};
csl_isa_field op{
op(){
set_width(2);
set_enum(e_op);
set_type(opcode);
}
};
csl_isa_element isa {
  isa(){
    set_type(root_format);
    set_width(32);
  }
};
csl_isa_element base_format: isa {
  e_op opcode;
  base_format() {
    set_type(inst_format);
    set_position(opcode, 30);
  }};
```

csl_isa_field *isa_field_object*(*width | bitrange*[,*enum*]);

DESCRIPTION :

Creates a ISA field *isa_field_object.*

param width - width of the field, it will transform into a range [n-1:0];

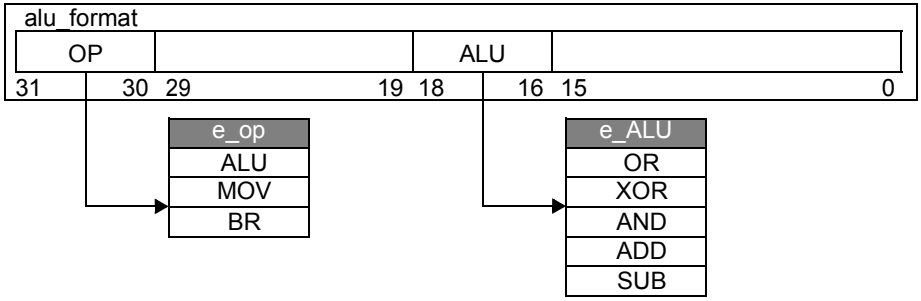
param enum_or_enum_item - the enum or the enum item will be associated to the field;

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

In this example rwo ISA fields *f_op* and *f_alu* are created and associated with different enums.

FIGURE 1.3



CSL CODE

```
csl_enum e_op{
 ALU,
 MOV,
 BR
};
csl_enum e_ALU{
 OR,
 XOR,
 AND,
 ADD,
 SUB
};
csl_isa_field op{
  op(){
  set_type(opcode);
  set_width(2);
  set_enum(e_op);
   }
};
csl_isa_field alu{
  alu(){
```

```
  set_type(selector);
  set_width(3);
  set_enum(e_ALU);
  }
};
csl_isa_element isa{
  isa(){
  set_type(root_format);
  set_width(32);
  }
};
csl_isa_element base_format : isa{
  op opcode;
  base_format(){
  set_type(instr_format);
  set_position(opcode,30);
  }
};
csl_isa_element alu_format : base_format{
  alu alu;
  alu_format(){
  set_type(instr_format);
  set_position(alu,16);
  }
};
```

VERILOG CODE

csl_isa_field *isa_field_object*(*field_object_name*);

DESCRIPTION :

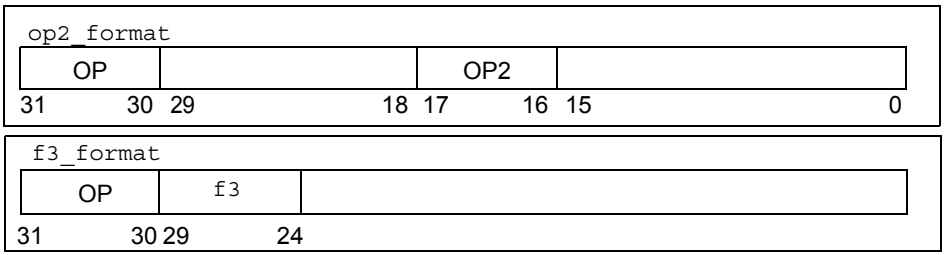
In this example an ISA field *isa_field_object* derived from another *csl_isa_field* is created.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

The ISA field *f_op2* is derived from ISA field *f_op.*

FIGURE 1.4



CSL CODE

```
csl_enum e_op{
   ADD,
   XOR,
   SUB
};
csl_isa_field op(2, e_op);
op.set_type(opcode);
csl_isa_field f3(24,29);
f3.set_type(address);
csl_isa_field op2(op);

csl_isa_format isa{
  isa(){
  set_type(root_format);
  set_width(32);
  }
};
csl_isa_format base_format : isa{
  op op1;
  base_format(){
  set_type(instr_format);
  set_position(op1,30);
  }
};
```

```
csl_isa_format op2_format : base_format{
  op2 op2;
  op2_format(){
  set_type(instr_format);
  set_position(op2,16);
  }
};
csl_isa_format f3_format : base_format{
  f3 field3;
  f3_format(){
  set_type(instr_format);
  set_position(field3,24);
  }
};
```

VERILOG CODE

csl_isa_field *isa_field_object*(*lower, upper*[,*enum*]);

DESCRIPTION :

Create an ISA field *isa_field_object*

params lower and upper - range of the field

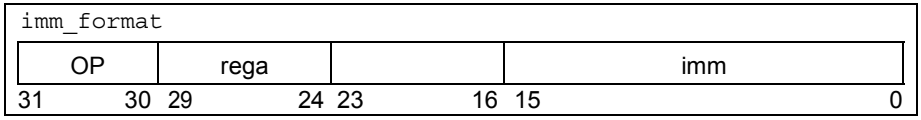
param enum_or_enum_item - the enum or the enum item will beassociated to the field;

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

In this example three ISA fields *f_op, f_rega, f_imm* are created using lower and upper range. The range [23:16] is not used and a child field may override this field with a range.

FIGURE 1.5



CSL CODE

```
csl_isa_field op(0,1);
op.set_type(opcode);

csl_isa_field rega(24,29);
rega.set_type(address);

csl_isa_field imm(0,15);
imm.set_type(constant);

csl_isa_format isa{
  isa(){
  set_type(root_format);
  set_width(32);
  }
};
csl_isa_format base_format : isa{
  op opcode;
  base_format(){
  set_type(instr_format);
  set_position(opcode,30);
  }
};
csl_isa_format reg_format : base_format{
  rega rega;
  reg_format(){
```

```
  set_type(instr_format);
  set_position(rega,24);
  }
};
csl_isa_format imm_format : reg_format{
  imm imm;
  imm_format(){
  set_type(instr_format);
  set_position(imm,0);
  }
};
```

VERILOG CODE

[*isa_field_obj_name.*]**set_type**(isa_field_type);

DESCRIPTION :

Set_type command is mandatory for a csl_isa_field. If set_type command is not set for a csl_isa_field then cslc will throw an error. A hierarchical isa_field can have as children fields only isa_fields. A hierarchical field can have as children fields only fields and not isa_fields;
The types an ISA field can have are detailed in Table 1.8

TABLE 1.8 ISA field types

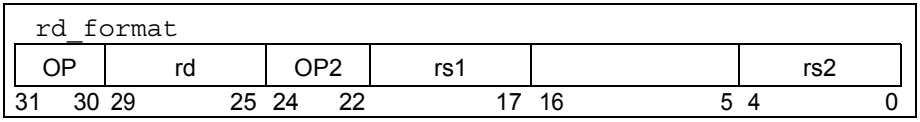
ISA_field_type	Details
opcode	opcode
subopcode	subopcode
rf_address	register file address
mem_address	memory address
branch_address	branch address
im	immediate constant
selector	selector
reserved	reserved by designer
constant	constant value
unused	unused

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the types of five ISA fields.

FIGURE 1.6



CSL CODE

```
csl_isa_field op(2);
op.set_type(opcode);

csl_isa_field op2(3);
op2.set_type(subopcode);

csl_isa_field rd(5);
rd.set_type(rf_address);

csl_isa_field rs1(5);
```

```
rs1.set_type(rf_address);

csl_isa_field rs2(5);
rs2.set_type(rf_address);

csl_isa_element isa{
  isa(){
  set_type(root_format);
  set_width(32);
  }
};
csl_isa_element base_format : isa{
  op opcode1;
  base_format(){
  set_type(instr_format);
  set_position(opcode1,30);
  }
};
csl_isa_element op2_format : base_format{
  op2 opcode2;
  op2_format(){
  set_type(instr_format);
  set_position(opcode2,22);
  }
};
csl_isa_element rd_format : op2_format{
  rd rd;
  rs1 rs1;
  rs2 rs2;
  rd_format(){
  set_type(instr_format);
  set_position(rd,25);
  set_position(rs1,17);
  set_position(rs2,0);
  }
};
```

VERILOG CODE

[*isa_field_obj_name*.]**set_name**(string);

DESCRIPTION :

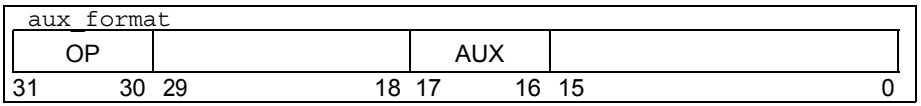
Sets the long name of the ISA_field that will be used when generating the pdf docs, if this is not set the default value is the name of the field ;

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the name “opcode” for the ISA field *f_op*.

FIGURE 1.7



CSL CODE

```
csl_isa_field op{
op(){
set_name("opcode");
set_type(opcode);
set_width(2);
}
};
csl_isa_field aux(2);
aux.set_type(selector);

csl_isa_element isa{
  isa(){
  set_type(root_format);
  set_width(32);
  }
};
csl_isa_element base_format : isa{
  op opcode;
  base_format(){
  set_type(instr_format);
  set_position(opcode,30);
  }
};
csl_isa_element aux_format : base_format{
  aux aux;
  aux_format(){
```

```
    set_type(instr_format);
    set_position(aux,16);
    }
};
```

VERILOG CODE

isa_field_obj_name.**set_mnemonic**(string);

DESCRIPTION :

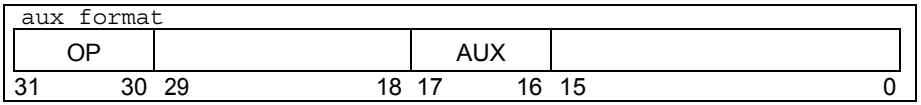
Sets the abrevation of the ISA_field that will be used when generating the c++ code, if this is not set the default value is the field name in small letters.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the abrevation "aux" of the ISA filed named *f_aux*.

FIGURE 1.8



CSL CODE

```
csl_isa_field op(2)
op.set_type(opcode);
csl_isa_field aux{
aux(){
set_type(selector);
set_mnemonic("aux");
set_width(2);
}
};
csl_isa_element isa{
  isa(){
  set_type(root_format);
  set_width(32);
  }
};
csl_isa_element base_format : isa{
  op opcode;
  base_format(){
  set_type(instr_format);
  set_position(opcode,30);
  }
};
csl_isa_element aux_format : base_format{
  aux aux;
  aux_format(){
  set_type(instr_format);
  set_position(aux,16);
```

```
    }
};
```

VERILOG CODE

```
isa_field_obj_name.set_offset(num_expr);
```

Description :

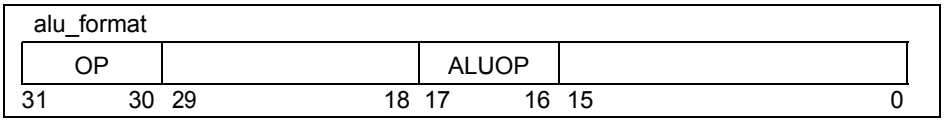
Set the value to be added to both lower and upper index of the ISA_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the offset for the ISA field *f_aluop*.

FIGURE 1.9



CSL CODE

```
csl_isa_field op(2,e_op);
csl_isa_field aluop{
aluop(){
set_width(2);
set_type(opcode);
set_offset(2);
}
};
csl_isa_element isa{
  isa(){
  set_type(root_format);
  set_width(32);
  }
};
csl_isa_element base_format : isa{
  op opcode;
  base_format(){
  set_type(instr_format);
  set_position(opcode,30);
  }
};
csl_isa_element alu_format : base_format{
  aluop aluop;
  alu_format(){
  set_type(instr_format);
  set_position(alu,16);
  }
};
```

VERILOG CODE

```
isa_field_obj_name.set_enum(enum_name);
```

DESCRIPTION :

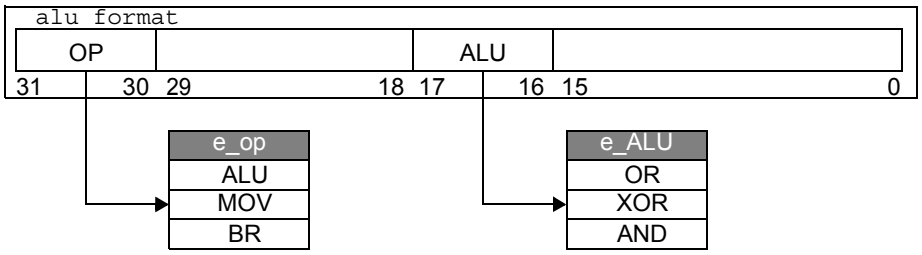
Associates an enum to the ISA field object.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Associates the enum *op_code* to the ISA field *op*.

FIGURE 1.10



CSL CODE

```
csl_enum op_code{
   ALU,
   MOV,
   BR
};
csl_enum eALU{
   OR,
   XOR,
   AND
};
csl_isa_field op{
op(){
set_type(opcode);
set_width(2);
set_enum(op_code);
}
};
csl_isa_field alu{
alu(){
set_type(selector);
set_width(2);
set_enum(eALU);
}
};
```

```
csl_isa_element isa{
  isa(){
  set_type(root_format);
  set_width(32);
  }
};
csl_isa_element base_format : isa{
  op opcode;
  base_format(){
  set_type(instr_format);
  set_position(opcode,30);
  }
};
csl_isa_element alu_format : base_format{
  alu alu;
  alu_format(){
  set_type(instr_format);
  set_position(alu,16);
  }
};
```

VERILOG CODE

```
isa_field_instance_obj_name.set_enum_item(enum_item_name);
```

DESCRIPTION :

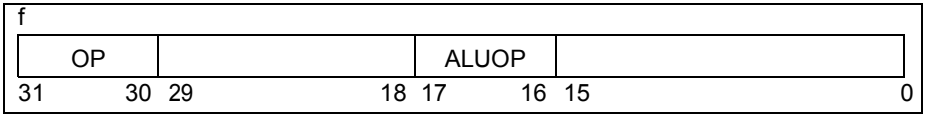
Associates an enum_item to the ISA field object, this is legal only for an instruction.

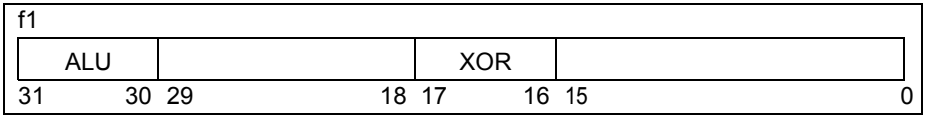
[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Associates an enum item *ALU* to the ISA field f*_op*, and an enum item *XOR* to ISA field *f_alu*.

FIGURE 1.11





CSL CODE

```
csl_enum e_op{
   ALU,
   MOV,
   BR
};
csl_enum e_ALUOP{
   XOR,
   OR,
   AND
};
csl_isa_field op(2, e_op);
op.set_type(opcode);

csl_isa_field aluop(2, e_ALU);
aluop.set_type(opcode);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element f: isa {
op op;
```

```
aluop aluop;
f() {
set_type(instr_format);
set_position(aluop, 16);
set_position(op, 30);
}
};
csl_isa_element f1: f{
f1() {
set_type(instr);
op.set_enum_item(ALU);
aluop.set_enum_item(XOR);
}
};
```

VERILOG CODE

```
isa_field_obj_name.set_value(num_expr);
```

DESCRIPTION :

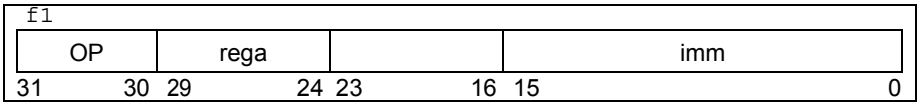
Associates a numeric value to the ISA_field object.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Associates the numeric value 244 with the ISA field *imm.*

FIGURE 1.12



CSL CODE

```
csl_isa_field op(2);
op.set_type(opcode);

csl_isa_field rega(6);
rega.set_type(address);

csl_isa_field imm{
imm(){
set_width(16);
set_type(constant);
set_value(244);
}
};
csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element f: isa {
op op;
f() {
set_type(instr_format);
set_position(op, 30);
}
};
csl_isa_element f1: f {
rega f_rega;
```

```
imm f_imm;
f1() {
set_type(instr_format);
set_position(f_rega, 24);
set_position(f_imm, 0);
}
};
```

VERILOG CODE

```
isa_field_obj_name.add_allowed_range(lower_num_expr,
upper_num_expr);
```

DESCRIPTION :

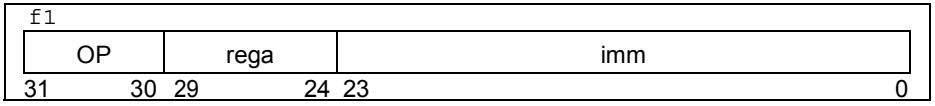
Sets the lower and upper bounds of the values that can be associated to the ISA field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Adds the allowed range for the ISA field *imm*.

FIGURE 1.13



CSL CODE

```
csl_isa_field op(2)
op.set_type(opcode);

csl_isa_field rega(6)
rega.set_type(address);

csl_isa_field imm{
  imm(){
  set_width(24);
  set_type(constant);
  add_allowed_range(10,156);
}
};
csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element f: isa {
op op;
f() {
set_type(instr_format);
set_position(op, 30);
}
};
csl_isa_element f1: f {
rega f_rega;
```

```
imm f_imm;
f1() {
set_type(instr_format);
set_position(f_rega, 24);
set_position(f_imm, 0);
}
};
```

VERILOG CODE

[signal/register.]**gen_decoder(csl_unit** *unit_object_name*);

DESCRIPTION :

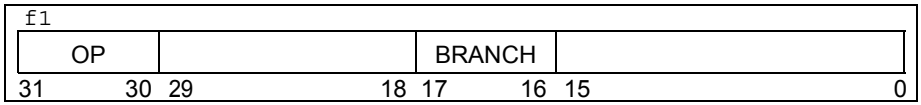
Generate a decoder for every field in the format. It is called in a csl_unit.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Generates a decoder for the unit *u*.

FIGURE 1.14



CSL CODE

```
csl_unit u{
  u(){}
};
csl_isa_field op(2);
op.set_type(opcode);
csl_isa_field branch{
  branch(){
  set_type(opcode);
  set_width(2);
  }
};
csl_isa_element isa {
  isa() {
  set_type(root_format);
  set_width(32);
  }
};
csl_isa_element f : isa {
  op f_op;
  f() {
  set_type(inst_format);
  set_position(f_op, 30);
  gen_decoder(u);
  }
};
csl_isa_element f1 : f{
  branch br;
  f1(){
```

```
    set_type(instr_format);
    set_position(br,16);
    }
  };
```

VERILOG CODE

csl_isa_element *isa_obj_name* [: *isa_obj_name0*];

DESCRIPTION :

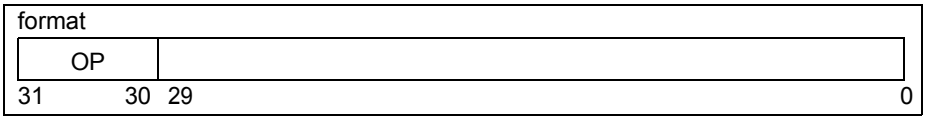
Csl_isa_element is a scope holder object. It can have instances and definitions of csl_isa_field and can be derived from another csl_isa_element .

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

//

FIGURE 1.15



CSL CODE

```
csl_isa_field op(2);
op.set_type(opcode);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element format : isa {
op f_op;
format() {
set_type(inst_format);
set_position(f_op, 30);
}
};
```

```
set_type (instr_format | instr | root_format | vliw_format );
```

DESCRIPTION :

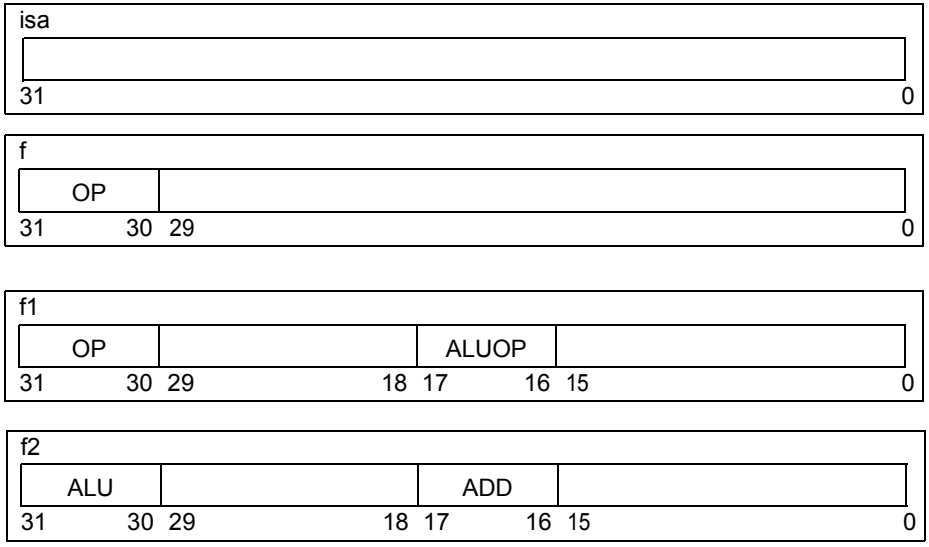
Set_type for csl_isa_element is mandatory. If the ISA_element is not derived from another ISA_element then the type of the element has to be root_format. A design can have multiple root_formats. A root_format cannot have ISA_field instantiations or definitions. The root_format name is the name that is considered as the ISA name.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the type "*root_format*" for ISA element *isa,* "*instr_format*" for ISA element *f* and "*instr*" for ISA element *f1*.

FIGURE 1.16



CSL CODE

```
csl_enum e_op{
   ALU,
   MOV
};
csl_enum e_aluop{
   ADD,
   SUB,
   XOR
};
csl_isa_field op(2, e_op);
op.set_type(opcode);
```

```
csl_isa_field aluop(2, e_aluop);
aluop.set_type(opcode);

csl_isa_element isa {
  isa() {
  set_type(root_format);
  set_width(32);
   }
};
csl_isa_element f: isa {
  op op;
  f() {
  set_type(instr_format);
  set_position(op, 30);
  }
};
csl_isa_element f1: f{
  aluop aluop;
  f1() {
  set_type(instr_format);
  set_position(aluop, 16);
  op.set_enum_item(ALU);
  }
};
csl_isa_element f2: f1{
  f2() {
  set_type(instr);
  aluop.set_enum_item(ADD);
  }
};
```

VERILOG CODE

```
set_width (numeric_expression);
```

DESCRIPTION :

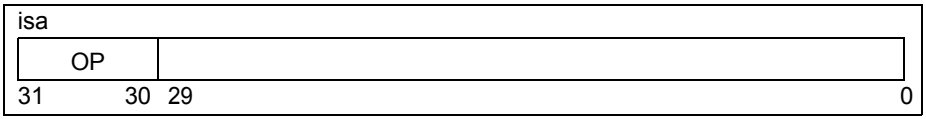
set_width for csl_isa_element is mandatory and can only be called for a root_format(set_type has to be before set_width);

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the width for the ISA element *isa*.

FIGURE 1.17



CSL CODE

```
csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
```

VERILOG CODE

set_position(*isa_field*, numeric_expression**);**

DESCRIPTION :

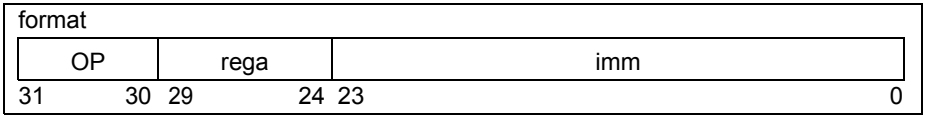
Set the absolute bit position of the ISA field, measured from the right side/0 bit position.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the position for ISA fields *op* and *imm.*

FIGURE 1.18



CSL CODE

```
csl_isa_field op(2);
op.set_type(selector);

csl_isa_field rega(6);
rega.set_type(address);

csl_isa_field imm(24);
imm.set_type(constant);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element format :isa {
op op;
rega rega;
imm imm;
format() {
set_type(instr_format);
set_position(op, 30);
set_next(op,rega);
set_position(imm,0);
}
};
```

VERILOG CODE

set_next(csl_isa_field left_*field*, **csl_isa_field** *right_field*);

DESCRIPTION :

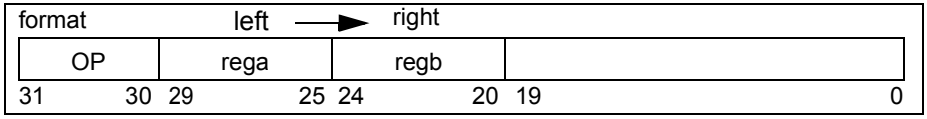
Set “the next “ field position of a field inside a format in a “linked-list“ way. In this way if the size of the fields changes they will remain adjacent to each other;

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Arranges the ISA fields *op,rega,regb*.

FIGURE 1.19 set_next



CSL CODE

```
csl_isa_field op(2);
op.set_type(selector);

csl_isa_field rega(5);
rega.set_type(address);

csl_isa_field regb(5);
regb.set_type(address);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element format : isa {
op op;
rega rega;
regb regb;
format() {
set_type(instr_format);
set_position(op, 30);
set_next(op,rega);
set_next(rega,regb);
}
};
```

VERILOG CODE

set_previous(csl_isa_field *left_field*, **csl_isa_field** *right_field*);

DESCRIPTION :

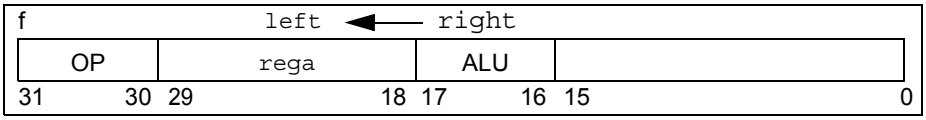
Set “the previous “ field position of a ISA field inside a format in a “linked-list“ way. In this way if the size of the ISA fields changes they will remain adjacent to each other.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

//

FIGURE 1.20 set_previous



CSL CODE

```
csl_isa_field op(2);
op.set_type(opcode);

csl_isa_file rega(12);
rega.set_type(address);

csl_isa_field alu(2);
alu.set_type(selector);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element f : isa{
op f_op;
rega f_rega;
alu f_alu;
f() {
set_type(instr_format);
set_position(f_alu, 16);
set_previous(f_alu,f_rega);
set_previous(f_rega,f_op);
}
};
```

VERILOG CODE

set_width (numeric_expression);

DESCRIPTION :

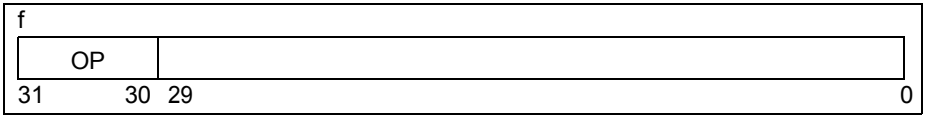
Set_width for csl_isa_field is mandatory

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

Sets the width for the ISA element *isa*.

FIGURE 1.21



CSL CODE

```
csl_isa_field op(2);
op.set_type(opcode);

csl_isa_element isa {
isa() {
set_type(root_format);
set_width(32);
}
};
csl_isa_element f : isa{
op f_op;
f() {
set_type(instr_format);
set_position(f_op, 16);
}
};
```

VERILOG CODE

set_range (lower_index,upper_index);

DESCRIPTION :

Sets the range for csl_isa_field.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_isa_field isa_fld{
  isa_fld(){
  set_range(0,4);
  }
};
```

set_bitrange (bitrange);

DESCRIPTION :

Sets the bitrange for csl_isa_field.

[*CSL ISA Syntax and Command Summary*]

EXAMPLE :

//

CSL CODE

```
csl_bitrange br(8);
csl_isa_field isa_fld{
  isa_fld(){
  set_bitrange(br);
  }
};
```

```
set_isa(isa_root_format_name);
```

DESCRIPTION :

An ISA can be associated with a register/signal. The condition is that the register/signal has the same width with isa_root_format. If the register/signal is declared without width or bitrange (default width 1), then the *set_isa*() command should change the register/signal bitrange to match the isa_root_element width.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

//

CSL CODE

```
csl_register_file rf_r{
  rf_r(){
  set_width(br_ir);
  }
};
csl_bitrange br_iw(0,31);
  csl_unit u{
  csl_signal sir(br_iw);
  rf_r rf_r;
  u(){
  sir.set_isa(isa_root_name);
  sir.gen_signals();
  rf_r.wr_addr=sir.base_format.rdst;
  rf_r.rd1_addr=sir.base_format.rsrc1;
  rf_r.rd2_addr=sir.base_format.rsrc2;
  }
};
```

VERILOG CODE

```
//
```

[register/signal.]**gen_signals();**

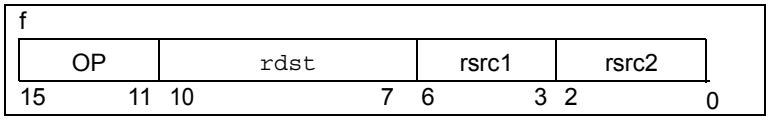
DESCRIPTION :

Generates signals corresponding to fields. Applies to register/signal that has associated an ISA. Generates a number of signals equal to number of fields in ISA. For each field there is a signal with the same width and name prefixed with [register/signal]_isa_name.

[***CSL ISA Syntax and Command Summary***]

EXAMPLE :

FIGURE 1.22 ISA Format 1



CSL CODE

```
csl_bitrange br_ir(16);
csl_register_file rf_r{
rf_r(){
set_width(br_ir);
}
};
csl_unit processor {
csl_signal sir(br_ir);
rf_r rf_r;
processor(){
sir.set_isa(isa_format_root_name);
sir.gen_signals();
rf_r.wr_addr=sir.base_format.rdst;
rf_r.rd1_addr=sir.base_format.rsrc1;
rf_r.rd2_addr=sir.base_format.rsrc2;
}};
```