



ADL Tools Documentation

Version 0.2

Table of Contents

1. Intro	1
2. Overview	2
3. What Is ADL-Tools Project?	3
4. Requirements and Installation	4
5. How to Run the Tools?	5
5.1. TD generation module	5
5.2. Testing module	6
5.2.1. Encoding tests	6
5.2.2. Relocation tests	7
6. How to Configure the Tools?	10
7. ADL XML – Architectural Description	21
8. How to generate register pairs?	32
9. How To Generate Intrinsic And Tests?	34
10. How to generate Sail description?	37
11. How to generate scheduling description?	40
12. Known Limitations	48

Chapter 1. Intro

This project is part of European project Tristan. Copyright 2024 NXP SPDX-License-Identifier: BSD-2-Clause

Chapter 2. Overview

This project comprises two main components aimed at facilitating the automatic generation and testing of LLVM target description files based on the information parsed from an architectural description of a RISC-V core.

- The first component focuses on the creation of target description files. These files serve as templates for generating instructions to be used within a specific context or application. The generation process involves defining various parameters, conditions, and constraints necessary for producing accurate and functional instructions.
- The second component is dedicated to generating encoding tests for the instructions within the target description files. This phase involves executing and evaluating the functionality and reliability of the instructions by rigorously validating each bit individually.

More information on ADL technology can be found at https://github.com/nxp-archive/adl-tools_adl/

Chapter 3. What Is ADL-Tools Project?

ADL-Tools is a tool suite designed for automatically generating target description files and encoding tests using an XML file. Practically, ADL-Tools parses an XML file, which contains an ADL model for one or more RISC-V extensions, and generates all the necessary .td files required by a compiler to integrate these new extensions. In simpler terms, the ADL represented as an XML file is crucial in this process, alongside the actual tools used.

Chapter 4. Requirements and Installation

- **Python 3.10** or higher (need to install python3 explicitly)
 - **numpy** package installed (python3 -m pip install numpy)
 - **num2words** package installed (python3 -m pip install num2words)
 - **word2number** package installed (python3 -m pip install word2number)
- **LLVM-LIT** (used for testing)
- **Asciidoctor** (browser extension for Release Notes)

NOTE | No installation is required; simply cloning the repository is sufficient.

NOTE | These modules were used and tested with **LLVM 18.x** version.

Chapter 5. How to Run the Tools?

5.1. TD generation module

In order to run the application, the user has to use **make_td.py** module which is the main module that triggers the other modules. Besides calling this module, the user also has to provide a proper XML file which is given as a command line argument when the module above is run. Finally, an example of running the application should look like:

```
python ./tools/make_td.py rv32i.adl.xml
```

Obviously, if the XML model is not in the same directory with the tools, then the user provides a path to the XML. In order to work properly, the user has to make sure the working directory is `tools_adl`, the main folder cloned.

Another feature provided is to select certain architecture from the XML given. For example, if the XML provides instructions for a bunch of extensions, the user can specify, using the command line arguments, which extensions should be generated. Beside specifying **the extensions enabled**, it is also possible to specify **the directory** in which the content is generated.

NOTE

If **no extension is specified**, then the tools generates content for **all the extensions declared** in the XML file, without exception.

NOTE

The user **must** be in **the root folder (tools_adl)** when running the script.

An example of selecting certain extensions and a specified directory for content generation is similar to:

```
python ./tools/make_td.py rv32i.adl.xml --extension=zclsd,zilsd,rv32i
--output=./examples/TD/test_zilsd
```

NOTE

Here the user selected to generate only the information related to `zclsd`, `zilsd` and `rv32i` extensions. Consequently, any other extension which is also defined in the `rv32i` XML model is ignored and nothing related to it generated. The flag in order to enable extension generation is **--extension=**, while for specifying the output directory is **--output=**. It is very important to pass the extensions **without any spaces, just commas (,)**.

NOTE

None of these 2 flags are mandatory, so any of these could not be specified. The flags are independent, consequently the user chooses if these is used or not. Moreover, it should consider the fact that the directory specified is **overwritten** at every run.

5.2. Testing module

5.2.1. Encoding tests

The Python script `make_test.py` is a tool for generating RISC-V instruction encoding tests based on an XML file containing extension information. It requires two mandatory arguments:

1. Path to XML file (`<path_to_xml_file>`): The location of the XML file that defines the RISC-V extensions.
2. Extensions (`<extensions>`): A comma-separated list of extensions (e.g., `rv32i`).

Additionally, an optional argument `-o` (or `--output`) can be used to specify the output directory where the tests are generated.

Usage

To run the script (`tools_adl/tools/testing/encoding/make_test.py`) use the following format:

```
python <rel_or_abs_path>/make_test.py <path_to_xml_file> <extensions> -o <output_path>
```

Example:

```
python make_test.py riscv_extensions.xml rv32i,rv32m -o encoding_tests
```

NOTE Run `"python make_test.py -h"` for detailed usage instructions.

Output Structure

Upon successful execution, the script generates two main outputs:

1. `info.py`: A Python file that contains information about instructions and operands.
2. Output directory: A folder named after the provided XML file (e.g., `adl.xml`) is created, containing the following subfolders:

a) `tests_<extensions>`: Contains instruction encoding tests for the specified extensions. These tests validate the correct encoding of each instruction.

b) `refs_<extensions>`: Contains reference encoding details for each instruction, which serve as a reliable source for comparison during the testing process.

Testing Environment Setup

After generating the tests, you can establish a testing environment using the `lit_references_tester.sh` script. The path to `lit.cfg` file is **`tools_adl/tools/testing/encoding/`**.

To set up the environment, run the following command:

```
source lit_references_tester.sh <path_to_llvm_asm> <path_to_llvm_readelf>
<path_to_tests_folder> <path_to_refs_folder> <path_to_lit_cfg>
```

Running Tests

Once the environment is set up, use `llvm-lit` to validate the generated instructions. The command format is as follows:

```
<path_to_llvm_lit>/llvm-lit --param app_asm=<path_to_llvm_asm> --param
app_readelf=<path_to_llvm_readelf> --param app_filecheck=<path_to_filecheck>
<path_to_tests_folder>
```

This command executes the tests and compares the generated instructions against the reference encodings, ensuring correct instruction behavior.

5.2.2. Relocation tests

NOTE

The user **must** run **make_test.py** script before running the relocations script in order to generate the **info.py** file.

The Python script `make_reloc.py` is a tool for generating RISC-V relocation tests based on an XML file containing extension information. It requires two mandatory arguments:

1. Path to XML file (`<path_to_xml_file>`): The location of the XML file that defines the RISC-V extensions.
2. Symbol table max value (`<symbol_max_value>`): Integer value for symbol table.

Additionally, an optional argument `-o` (or `--output`) can be used to specify the output directory where the tests are generated.

Usage

To run the script (`tools_adl/tools/testing/relocations/make_reloc.py`) use the following format:

```
python <rel_or_abs_path>/make_reloc.py <path_to_xml_file> <symbol_max_value> -o
<output_path>
```

Example:

```
python make_reloc.py riscv_extensions.xml 10 -o relocation_tests
```

NOTE

Run `"python make_reloc.py -h"` for detailed usage instructions.

Output Structure

Upon successful execution, the script generates an output directory named after the provided XML file (e.g., adl.xml), containing the following subfolders:

a) tests: Contains relocation tests for the previously ran instruction tests using `*make_test.py*` module. These tests validate the correct generation of each relocation.

b) refs: Contains reference details for each relocation, which serve as a reliable source for comparison during the testing process.

Running Tests

You can establish a testing environment by copying the `lit.cfg` file from `tools_adl/tools/testing/relocations` inside the output folder (e.g. `relocation_tests` from the example above).

Once the environment is set up, use `llvm-lit` to validate the generated relocations. The command format is as follows:

```
<path_to_llvm_lit>/llvm-lit --param app_asm=<path_to_llvm_asm> --param  
app_readelf=<path_to_llvm_readelf> --param app_filecheck=<path_to_filecheck>  
<path_to_reloc_tests_folder>
```

This command executes the tests and compares the generated relocations against the reference outputs, ensuring correct relocation behavior.

- Generated relocations table:

Value	Relocation	Instrfield
1	R_RISCV_32	N/A
2	R_RISCV_64	N/A
20	R_RISCV_GOT_HI20	imm_u_pc
21	R_RISCV_TLS_GOT_HI20	imm_u_pc
22	R_RISCV_TLS_GD_HI20	imm_u_pc
23	R_RISCV_PCREL_HI20	imm_u_pc
24	R_RISCV_PCREL_LO12_I	imm_i
25	R_RISCV_PCREL_LO12_S	imm_s
26	R_RISCV_HI20	imm_u
27	R_RISCV_LO12_I	imm_i
28	R_RISCV_LO12_S	imm_s

Value	Relocation	Instrfield
29	R_RISCV_TPREL_HI20	imm_u
30	R_RISCV_TPREL_LO12_I	imm_i
31	R_RISCV_TPREL_LO12_S	imm_s
32	R_RISCV_TPREL_ADD	N/A
33	R_RISCV_ADD8	N/A
34	R_RISCV_ADD16	N/A
35	R_RISCV_ADD32	N/A
36	R_RISCV_ADD64	N/A
37	R_RISCV_SUB8	N/A
38	R_RISCV_SUB16	N/A
39	R_RISCV_SUB32	N/A
40	R_RISCV_SUB64	N/A
54	R_RISCV_SET8	N/A
55	R_RISCV_SET16	N/A
56	R_RISCV_SET32	N/A

Chapter 6. How to Configure the Tools?

For configure and add/remove additional information, there are 2 files which are used for this kind of actions (**llvm-config.txt** and **config.txt**). As the names of these files suggest, the first one is used to define additional information related to the LLVM layout or related to the .td files generated. Generally, all the definitions represent information which the tools cannot find in the XML parsed but which is essential for a complete and correct generation. The second file is shorter and it should not be changed by the user, unless there are some important reasons to do so.

In order to understand the content and structure of llvm-config.txt, we analyze the sections and content of this file.

- **TD files used for generation**

- This section contains information about the .td files generated.
- It specifies which are the .td files generated and which is the name the .td file has. The user defines for each field listed below the path or the folder (depending on the case) where the content is generated. The name given to a certain td file should preserve LLVM format name, similar to those listed below.

```
RegisterInfoFile = RISCVRegisterInfo_gen.td
```

```
InstructionInfoFile = RISCVInstrInfo_gen.td
```

```
InstructionFormatFile = RISCVInstrFormats_gen.td
```

```
InstructionFormatFile16 = RISCVInstrFormats16_gen.td
```

```
InstructionAliases = RISCVInstrAliases_gen.td
```

```
OperandsFile = RISCVOperands_gen.td
```

```
OperandsFile16 = RISCVOperands16_gen.td
```

```
CallingConventionFile = RISCVCallingConv_gen.td
```

```
RelocationFile = RISCVReloc.def
```

```
IntrinsicsFile = RISCVIntrinsics_gen.td
```

```
BuiltinFile = BuiltinRISCV.def
```

```
BuiltinHeader = riscv_builtin.h
```

```
MemoryOperand = RISCMemoryOperand_gen.td
```

```
TestIntrinsics = Tests
```

- Left value is a variable which represents the identifier for the Instruction .td file, while the right value is the name given to the Instruction file which can be changed.

- **LLVM Configuration Variables and Setup**

- This section includes information about environment variables or other variables needed for setup. All information is related to the LLVM standard requirements (information about register classes, constraints, debug info, instructions width etc). The user generally does not change this section unless the information to change is mandatory. The structure is the same as it was for the section presented before. The left value should not be edited, while the right value may be changed.

```
Namespace = RISCV
```

```
BaseArchitecture = rv32
```

- The user can define a certain register class.

```
RegisterClass = RISCVReg
```

- The user can enable subregister generation if necessary.

```
RegisterClassSubRegs_GPR = RISCVRegWithSubRegs
```

- The user can define instruction classes and formats.

```
InstructionClass = RVInst
```

```
InstructionClassC = RVInst16
```

```
InstructionFormat = InstFormat
```

- The user can define ABI information.

```
RegAltNameIndex = ABIRegAltName
```

- The user can set register and instruction width.

```
LLVMGPRBasicWidth = 32
```

```
LLVMStandardInstructionWidth = 32
```

- The user can set several LLVM information which are used in the script.

```
AsmString = opcodestr # "\t" # argstr
```

```
LLVMConstraintClassWidth = 3
```

```
LLVMConstraintRiscVPrefix = RV
```

```
LLVMConstraintName = VConstraint
```

```
LLVMConstraintValues = NoConstraint
```

```
LLVMNoConstraintValue = 0b000
```

```
TSFlagsFirstConstraint = 7
```

```
TSFlagsLastConstraint = 5
```

- The user can define sideEffect attributes and memory synchronization attribute

```
sideEffectAttributeSpecific = sideEffect
```

```
memorySynchronizationInstruction = sync
```

- The user can set XLenVT and XLenRI information used in LLVM.

```
XLenVT = i32
```

```
XLenVT_key = XLenVT
```

```
XLenRI = RegInfo<32,32,32>
```

```
XLenRI_key = XLenRI
```

- The user can enable SP generation

```
DefineSP = True
```

• Instructions Types

- This section contains instruction types definitions. Based on the attributes defined in the XML model, there are several instructions types: branch, store, load, jump etc. Moreover, for compressed instructions, there a few instruction types defined. The left value is standard and should not be changed, while the right value may be changed, but it should preserve the same format as for those listed below (InstrFormat):

```
instructionFormatR = InstFormatR
```

```
instructionFormatCR = InstFormatCR
```

```
instructionFormatI = InstFormatI
```

```
instructionFormatCI = InstFormatCI
```

```
instructionFormatB = InstFormatB
```

```
instructionFormatCB = InstFormatCB
```

```
instructionFormatJ = InstFormatJ
```

```
instructionFormatU = InstFormatU
```

```
instructionFormatS = InstFormatS
```

```
instructionFormatCS = InstFormatCS
```

• LLVM Format Info

- This section describes the LLVM format, containing all the information needed for LLVM Instruction Format generation. It specifies which are TSFlags fields and also contains information about TSFlags definitions, specifies ImmAsmOperands classes and parameters and other information required by LLVM layout.
- The user can set aliases for GPR subclasses. The value after _ is the offset for the register subclass.

```
aliasGPR_8 = GPRC
```

```
aliasGPR_1 = GPRNoX0
```

```
aliasGPR_1Nox2 = GPRNoX0X2
```

- The user can set several information specific to LLVM format

```
LLVMPrivilegedAttributes = {rv32pa}
```

```
LLVMOtherVTAttrib = {branch}
```

```
LLVMOtherVTReloc = {}
```

```
LLVMOtherVTValue = OtherVT
```



```
LLVMPrintMethodAttrib = {branch}
```

```
LLVMPrintMethodReloc = {}
```

```
LLVMPrintMethodValue = printBranchOperand
```

```
LLVMOperandTypeAttrib = {branch}
```

```
LLVMOperandTypeReloc = {}
```

```
LLVMOperandTypeValue = OPERAND_PCREL
```

- The user can provide information about LLVM Operand Class format

```
SImmAsmOperandParameters = {int_width, string_suffix}
```

```
UImmAsmOperandParameters = {int_width, string_suffix}
```

```
ImmAsmOperandParameters = {string_prefix, int_width, string_suffix}
```

```
ImmAsmOperandName = {prefix, width, suffix}
```

```
ImmAsmOperandRenderMethod = addImmOperands
```

```
ImmAsmOperandDiagnosticType = !strconcat("Invalid", Name)
```

```
basicDecodeMethod = {decodeUImmOperand, decodeSImmOperand}
```

- The user should set the information for LLVM Flags. The user could change the values based on the LLVM version or if a known change is required.

```
TSFlagsFirst = 4
```

TSFlagsLast = 0

LLVMVFlags = {VLMul, HasDummyMask, ForceTailAgnostic, HasMergeOp, HasSEWOp,
HasVLOp, HasVecPolicyOp, IsRVWideningReduction, UsesMaskPolicy,
IsSignExtendingOpW}

VLMul = 0

VLmulTSFlagsStart = 10

VLmulTSFlagsEnd = 8

HasDummyMask = 0

HasDummyMaskTSFlagsStart = 11

HasDummyMaskTSFlagsEnd = 11

ForceTailAgnostic = false

ForceTailAgnosticTSFlagsStart = 12

ForceTailAgnosticTSFlagsEnd = 12

HasMergeOp = 0

HasMergeOpTSFlagsStart = 13

HasMergeOpTSFlagsEnd = 13

HasSEWOp = 0

HasSEWOpTSFlagsStart = 14

HasSEWOpTSFlagsEnd = 14

HasVL0p = 0

HasVL0pTSFlagsStart = 15

HasVL0pTSFlagsEnd = 15

HasVecPolicyOp = 0

HasVecPolicyOpTSFlagsStart = 16

HasVecPolicyOpTSFlagsEnd = 16

IsRVVWideningReduction = 0

IsRVVWideningReductionTSFlagsStart = 17

IsRVVWideningReductionTSFlagsEnd = 17

UsesMaskPolicy = 0

UsesMaskPolicyTSFlagsStart = 18

UsesMaskPolicyTSFlagsEnd = 18

IsSignExtendingOpW = 0

```
IsSignExtendingOpWTSFlagsStart = 19
```

```
IsSignExtendingOpWTSFlagsEnd = 19
```

- **Calling Convention**

- This sections contains calling convention information. It specifies the calling convention policy. RegisterAllocationOrder is a dictionary in which the keys represent the register classes and the values are lists specifying the calling convention allocation order. The other entries from this sections specifies additional information.

```
RegisterAllocationOrder = {GPR: [Function_arguments, Temporary, Saved_register,  
Hard_wired_zero, Return_address, Stack_pointer, Global_pointer, Thread_pointer]}
```

- The user can define calling convention allocation order

```
CallingConventionAllocationOrder = {CSR_ILP32_LP64: [Return_address,  
Global_pointer, Thread_pointer, Saved_register]}
```

```
CallingConventionAllocationExcluded = {CSR_Interrupt: [Hard_wired_zero,  
Stack_pointer]}
```

```
CSR_ILP32_LP64_Ref = GPR
```

```
CSR_Interrupt_Ref = GPR
```

- The user can set other XLenRI and XLenVT information

```
XLenRIRegInfo = RegInfoByHwMode<[RV32, RV64], [RegInfo<32,32,32>,  
RegInfo<64,64,64>]>
```

```
XLenVTValueType = ValueTypeByHwMode<[RV32, RV64], [i32, i64]>
```

- **Extensions Declaration**

- This section declares the extensions that are generated if they are found in the XML model. In other words, if an extension is used or it should be generated, then it has to be defined in this section, otherwise it is ignored, even if they are found in the XML model.

```
LLVMExtRv32test = HasStdExtRV32Test
```

```
HasStdExtRV32TestExtension = RV32Test
```

- The first line declared specify the attribute that is found in the XML model for each instruction that belongs to this extension. Basically, **RV32Test** is the attribute for a test extension so the left value is built by appending the attribute **Rv32Test** capitalized to the **LLVMExt** keyword. **RV32Test** attribute represents in fact the **RV32Test** extension, so the right value is built by appending **RV32Test** to **HasStdExt** keyword.
- The second line declared is built by appending the previous extension **RV32Test** to **HasStdExt** keyword and then Extension suffix is added to this structure. The right value is the extension itself **RV32Test**.

- **Immediate Operands**

- This section declares the immediate operands that have special declarations which can not be automatically generated with the information found in the XML model. Firstly, ImmediateOperands is a list in which the used should specify an operand which has a special declaration. After that, the same operand becomes an entry in this section, building a kind of dictionary. For this operand, the user defines between \{} the components that are used for generation such as: AliasImmClass which is an alias that is used instead of the basic name for the operand, ParserMatchClass, PrintMethod etc. If an operand is now defined here, then it is generated using only the information found in the XML model used, so the content could be incorrect or incomplete.

```
ImmediateOperands = {immu_ci, fence_prod, fence_succ,  
GenericOperand, imm_cbdnez, imm_uj, shamt_c, imm_u_pc, imm_u, imm_sb,  
pd, ps1, ps2, ps3, s1_ptr, d_ptr, imm_send, rm}
```

```
immu_ci = {AliasImmClass=c_lui_imm, DefineOperand=CLUIImmAsmOperand,  
ParserMatchClass=CLUIImmAsmOperand, ImmAsmOperandName=CLUIImm,  
ImmAsmOperandRenderMethod=addImmOperands,  
ImmAsmOperandDiagnosticType=!strconcat("Invalid", Name),  
DecoderMethod=decodeCLUIImmOperand, OperandClass=AsmOperandClass}
```

- **Additional Extensions Info**

- This section contains additional information for certain extensions. It could specify for example if certain extensions should have a prefix for the instructions or if there are special DecoderNamespace values. For a default case, the DecoderNamespace defined is "RV32Only_".

```
DecoderNamespace = {Others=RISCV32Only_}
```

Beside the configuration file, the user should add some important information in the XML model.

- **Instruction field definition**

Firstly, the user should provide create new instruction fields for special register subclasses. For

example, if the user needs to define a special subregister class such as **GPRC**, it should be a new instruction field defined in the XML file which has a reference to the parent register class, in this case **GPR**. The instruction field should be similar to other instruction field already defined.

- **Change of flow and other additional attributes**

The user should add change of flow attributes for specific instruction such as **branch**, **jumps** or **other type** of instructions. The attributes supported for these types of instructions are:

- **branch**
- **jump**
- **u-type**

For the instruction having **Side Effects** or **Memory Synchronization**, the user should also add in the XML the attributes needed depending on case:

- **sideEffect**
- **sync**
- **Excluded Values and Sign Extension**

The user should add **<sign_extension>** information for the instruction's operands which asks for. Moreover, the user should fill **<excluded_values>** field with information for specifying if any value should be not used.

```
<excluded_values>
  <option name="rdx">
    <str>x2</str>
  </option>
  <option name="rdx">
    <str>sp</str>
  </option>
</excluded_values>
```

```
<sign_extension>
  <int>20</int>
</sign_extension>
```

Chapter 7. ADL XML – Architectural Description

This document describes the ADL xml layout. This file is meant to provide more details about the tags and sub-tags found in an ADL xml file, how these tags are used, but also specifies if these are mandatory or not for a proper usage of the tools. The file contains all the tags and sub-tags found in **RV32I.adl.xml** model.

<data>

- **<cores>**
 - **<core>** - Information about the core and architecture for which the xml is written
 - **<doc>** (str) - Documentation
 - **<bit_endianness>** (str) – Endianness type
 - **<type_declaration>** (str) - The enum values may then be used within action code, or to initialize field values, such as cache, MMU, or event-bus fields
 - **<RaMask>** - Specify a real-address mask. This is applied to all addresses after translation, but before the request to memory.
 - **<initial>** (str) – Specify the address
 - **<constant>** (str) – True/false value
 - **<EaMask>** - Specify an effective-address mask. This is applied to all addresses immediately before translation.
 - **<initial>** (str) – Specify the address
 - **<constant>** (str) – True/false value
 - **<regs>**
 - **<register name⇒>** (str) – A valid C++ identifier
 - **<doc>** (str) – Documentation
 - **<width>** (int) - Specifies the register width in bits.
 - **<attributes>** - Lists any attributes that this register is associated with.
 - **<attribute name⇒>** (str) - A valid string identifier
 - **<str>** – Optional value given to the attribute
 - **<reset>** (str) – The reset value or text of the function called to reset the register
 - **<shared>** (int) – 1 or 0. Non-zero implies that the register is shared by other cores in the system.

<regs>	Data type	Occurrence	Usage	Child tags	Parent tags
<register name>	str	Mandatory	Used	-	-

<regs>	Data type	Occurrence	Usage	Child tags	Parent tags
<doc>	str	Optional	Not used	-	<register name>
<width>	int	Mandatory	Used	-	<register name>
<attributes>	str	Mandatory	Used	<attribute name>	<register name>
<attribute name>	str	Mandatory	Used	-	<attributes>
<shared>	int	Optional	Not used	-	<register name>
<reset>	str	Optional	Not used	-	<register name>

- **<regfiles>**

- **<regfile name>**⇒ (str) – Define a register file. This basically follows the format of a register. The register name must be a valid C++ identifier and may be referred to within action code by using its name
 - **<doc>** (str) – Documentation
 - **<width>** (int) – Same as for reg
 - **<attributes>** – Same as for reg
 - **<attribute name>**⇒ (str) - A string identifier
 - **<str>** – Optional value given to the attribute
 - **<size>** (int) – The number of entries in the register file.
 - **<debug>** (int) – Used for storing debug information
 - **<shared>** (int) – 1 or 0. Non-zero implies that the register is shared by other cores in the system.
 - **<calling_convention>** – A list used for specifying calling convention information.
 - **<option name>**⇒ (str) - String identifier for option
 - **<entries>** – A list containing all the entries for a register file. It has to match the options listed in **<enumerated>** tag from the instruction fields associated.
 - **<entry name>**⇒ (str) – Name given to the entry
 - **<syntax>** (str) – other name associated **<read>** (str) – read actions
 - **<write>** (str) – write actions

<regfiles>	Data type	Occurrence	Usage	Child tags	Parent tags
<regfile name>	str	Mandatory	Used	-	<regfile name>

<regfiles>	Data type	Occurrence	Usage	Child tags	Parent tags
<doc>	str	Optional	Not used	-	<regfile name>
<width>	int	Mandatory	Used	-	<regfile name>
<attributes>	str	Mandatory	Used	<attribute name>	<regfile name>
<attribute name>	str	Mandatory	Used	-	<attributes>
<size>	int	Mandatory	Used	-	<regfile name>
<debug>	int	Optional	Used	-	<regfile name>
<shared>	int	Optional	Not used	-	-
<calling_convention>	str	Optional	Used	<option name>	<regfile name>
<option name>	str	Optional	Used	-	<calling_convention>
<entries>	str	Mandatory	Used	-	<regfile name>
<entry name>	str	Mandatory	Used	-	<entries>
<syntax>	str	Mandatory	Used	-	<regfile name>
<read>	str	Mandatory	Not used	-	<regfile name>
<write>	str	Mandatory	Not used	-	<regfile name>

- **<relocations>**

- **<reloc name>** ⇒ (str) – Define a linker relocation type. A relocation is the method by which an assembler communicates with a linker, when symbol addresses cannot be determined at assembly time.
 - **<abbrev>** (str) – Optional abbreviation used within the assembly file. If not specified, then the relocation's name is used instead.
 - **<field_width>** (int) – Width of field used with this relocation, in bits. If a width is specified and it is also used by an instruction field, then the widths must match.
 - **<pcrel>** (str) – Optional, whether or not this is a pc-relative relocation.
 - **<value>** (int) – Integer value of the relocation.
 - **<right_shift>** (int) – Optional, used to specify the number of bits the relocation value is right-shifted before it is encoded.
 - **<dependency>** (str) – Optional, handles the high part of the relocation, helping manage memory offset.

<relocations>	Data type	Occurrence	Usage	Child tags	Parent tags
<reloc name>	str	Mandatory	Used	-	-

<relocations>	Data type	Occurrence	Usage	Child tags	Parent tags
<abrev>	str	Optional	Used (testing)	-	<reloc name>
<field_width>	int	Optional	Not used	-	<reloc name>
<pcrel>	str	Optional	Not used	-	<reloc name>
<value>	int	Mandatory	Used	-	<reloc name>
<right_shift>	int	Optional	Not used	-	<reloc name>
<dependency>	str	Optional	Used (testing)	-	<reloc name>

- **<instrfields>**

- **<instrfield name>** ⇒ (str) – Define an instruction field.
 - **<doc>** (str) – Documentation
 - **<bits>** – A list of integers representing the bit indices
 - **<range>** (int) – Valid ranges.
 - **<width>** (int) – Field width, in bits
 - **<size>** (int) – Field computed value, in bits.
 - **<shift>** (int) – Specify a shift value for the field. Within an instruction's action code, the value for the field is the field's encoded value shifted left by the specified number of bits.
 - **<offset>** (int) – Specify an implicit offset. Within an instruction's action code, the value for the field is the field's encoded value plus the offset.
 - **<mask>** (str) – specify an allowed mask
 - **<type>** (str) – Specifies the type of this instruction field.(regfile, imm)
 - **<enumerated>** – A list containing the entries for the instruction field. It has to match the <entries> tag for the <regfile> associated if applicable.
 - **<option name>** ⇒ (str) - String identifier for option
 - **<ref>** (str) – If the type is one which refers to another resource, such as *regfile*, *memory*, or *instr*, this key specifies the association.
 - **<signed>** (str) – If an immediate field, this specifies whether it is a signed quantity.
 - **<reloc>** (str) – specify the reocation associated
 - **<unsigned_upper_bound>** (str) – If a signed immediate field, then this specifies that the allowed upper bound should be treated as an unsigned number, when performing range checking, such as by the assembler.

<instrfields>	Data type	Occurrence	Usage	Child tags	Parent tags
<instrfield name>	str	Mandatory	Used	-	-

<instrfields>	Data type	Occurrence	Usage	Child tags	Parent tags
<doc>	str	Optional	Not used	-	<instrfield name>
<bits>	-	Mandatory	Used	<range>	<instrfield name>
<range>	int	Mandatory	Used	-	<bits>
<width>	int	Mandatory	Used	-	<instrfield name>
<size>	int	Mandatory	Used	-	<instrfield name>
<offset>	int	Mandatory	Used	-	<instrfield name>
<mask>	str	Mandatory	Not used	-	<instrfield name>
<type>	str	Mandatory	Used	-	<instrfield name>
<enumerated>	-	Mandatory	Used	<option name>	<instrfield name>
<option name>	str	Mandatory	Used	-	<enumerated>
<ref>	str	Mandatory	Used	-	<instrfield name>
<signed>	str	Mandatory	Used	-	<instrfield name>
<reloc>	str	Optional	Not used	-	<instrfield name>
<unsigned_upper_bound>	str	Optional	Not used	-	<instrfield name>

- **<instrs>**

- **<instruction name>** ⇒ (str) – Define an instruction.
 - **<width>** (int) – Instruction width, in bits.
 - **<doc>** (str) – Documentation
 - **<syntax>** (str) – Specifies how an instruction is to be parsed by an assembler or printed by a disassembler.
 - **<dsyntax>** (str) – Specifies how an instruction is to be printed by a disassembler.
 - **<attributes>** – Lists any attributes that this instruction is associated with.
 - **<attribute name>** ⇒ (str) - String identifier
 - <str> – Optional value given to the attribute
 - **<fields>** – A list of fields, sub-instructions, or bit-mapped fields.
 - **<field name>** ⇒ (str) - String identifier for field
 - **<action>** (str) – The semantics of the instruction. Instruction fields are accessible using their names and registers are also accessible using their names.
 - **<disassemble>** (str) – This is a hint which tells ADL whether to exclude this instruction when attempting to disassemble an opcode.
 - **<inputs>** (str) – a list containing all the fields that are read
 - **<outputs>** (str) – a list containing all the fields that are written

- **<intrinsic>** (str) – Tag used for specifying the intrinsic
- **<intrinsic_args>** (str) – Tag used for specifying the intrinsic arguments
- **<intrinsic_type>** – List used for defining arguments types for intrinsic
 - **<instrfield_intrinsic name⇒>** (str) - String name identifier
 - **<str>** – Intrinsic type
- **<generate_builtin>** (str) – Tag used for specifying information about builtin generation
- **<aliases>** – The function name (or names) must be that of another instruction already defined.
 - **<alias name⇒>** (str) – The name given to the alias
 - **<sources>** – specify the sources read when used
 - **<source>**
 - **<field>** (str) – specify the field read which takes a certain value
 - **<value>** (int) – specify the value
 - **<destinations>** – specify the destinations written when used
 - **<destination>**
 - **<field>** (str) – specify the field written which takes a certain value
 - **<value>** (int) – specify the value
 - **<parent_action>** (str) – specify the action done by the instruction for which alias is defined
 - **<excluded_values>** – List which specifies if a value should be avoided when defining or using
 - **<option name⇒>** (str) - Option string identifier
 - **<int>** – Excluded value
- **<helpers>** (str) – List any core-level helper functions used by the instruction.
- **<raises_exceptions>** (str) – If true, the instruction may raise an explicit exception.

<instrs>	Data type	Occurrence	Usage	Child tags	Parent tags
<instruction name>	str	Mandatory	Used	-	-
<doc>	str	Optional	Not used	-	<instruction name>
<width>	int	Mandatory	Used	-	<instruction name>
<syntax>	str	Mandatory	Used	-	<instruction name>
<dsyntax>	str	Mandatory	Used	-	<instruction name>

<instrs>	Data type	Occurrence	Usage	Child tags	Parent tags
<attributes>	str	Mandatory	Used	<attribute name>	<instruction name>
<attribute name>	str	Mandatory	Used	-	<attributes>
<fields>	-	Mandatory	Used	<field name>	<instruction name>
<field name>	str	Mandatory	Used	-	<fields>
<action>	str	Mandatory	Used	-	<instruction name>
<disassemble>	str	Optional	Not used	-	<instruction name>
<inputs>	str	Mandatory	Used	-	<instruction name>
<outputs>	str	Optional	Used	-	<instruction name>
<intrinsic>	str	Optional	Used	-	<instruction name>
<intrinsic_args>	str	Mandatory	Used	-	<instruction name>
<intrinsic_type>	-	Mandatory	Used	-	<instruction name>
<instrfield_intrinsic_name>	str	Mandatory	Used	-	<instruction name>
<generate_builtin>	str	Mandatory	Used	-	<instruction name>
<aliases>	-	Optional	Used	<alias name>	<instruction name>
<alias name>	str	Mandatory	Used	-	<aliases>
<sources>	-	Mandatory	Used	<source>	<aliases>
<source>	-	Mandatory	Used	<field>, <value>	<sources>
<field>	str	Mandatory	Used	-	<source>
<value>	int	Mandatory	Used	-	<source>
<destinations>	-	Mandatory	Used	<destination>	<aliases>
<destination>	-	Mandatory	Used	<field>, <value>	<destinations>
<field>	str	Mandatory	Used	-	<destination>

<instrs>	Data type	Occurrence	Usage	Child tags	Parent tags
<value>	int	Mandatory	Used	-	<destination>
<parent_action>	str	Mandatory	Used	-	-
<excluded_values>	-	Optional	Used	<option name>	-
<option name>	str	Mandatory	Used	-	<excluded values>
<helpers>	str	Optional	Not used	-	-
<raises_exceptions>	str	Optional	Not used	-	-

- **<exceptions>**

- **<exception name>** ⇒ (str) – Define an exception. Exception names must be valid C++ identifiers
 - **<doc>** (str) – Documentation
 - **<priority>** (str) – Specifies the priority class for the exception.
 - **<action>** (str) – This code is executed when the exception is raised.

<exceptions>	Data type	Occurrence	Usage	Child tags	Parent tags
<exception name>	str	Optional	Not used	-	-
<doc>	str	Optional	Not used	-	<exception name>
<priority>	str	Optional	Not Used	-	<exception name>
<action>	str	Optional	Not Used	-	<exception name>

- **<core-level-hooks>** - Lists various hook functions associated with the core.

- **<decode-miss>** (str) – Code to be executed on a decode miss.
- **<pre-cycle>** (str) – Code to be executed once per cycle, at the beginning of the cycle.
- **<post-cycle>** (str) – Code to be executed once per cycle, at the end of the cycle.
- **<pre-pre-fetch>** (str) –
- **<pre-fetch>** (str) – Code to be executed immediately before an instruction fetch.

- **<post-fetch>** (str) – Code to be executed immediately after an instruction fetch.
- **<post-exec>** (str) – Code to be executed immediately after an instruction has been executed.
- **<post-asm>** (str) – Code to be executed by the assembler immediately after an instruction has been assembled from its operands.
- **<post-packet-asm>** (str) – Code to be executed by the assembler after a packet of instructions has been assembled.
- **<post-packet>** (str) – Code to be executed after a packet of instructions has been executed.
- **<active-watch>** (str) – Predicate to determine if the core is currently active or halted.
- **<instr-table-watch>** (str) – Code which determines the current instruction table currently in effect.

<core_level_hooks>	Data type	Occurrence	Usage	Child tags	Parent tags
<decode_miss>	str	Optional	Not used	-	<core_level_hooks>
<pre_cycle>	str	Optional	Not used	-	<core_level_hooks>
<post_cycle>	str	Optional	Not used	-	<core_level_hooks>
<pre-pre-fetch>	str	Optional	Not used	-	<core_level_hooks>
<pre-fetch>	str	Optional	Not used	-	<core_level_hooks>
<post-fetch>	str	Optional	Not used	-	<core_level_hooks>
<post-exec>	str	Optional	Not used	-	<core_level_hooks>
<post-asm>	str	Optional	Not used	-	<core_level_hooks>
<post-packet-asm>	str	Optional	Not used	-	<core_level_hooks>
<post-packet>	str	Optional	Not used	-	<core_level_hooks>
<active-watch>	Str	Optional	Not used	-	<core_level_hooks>
<instr-table-watch>	str	Optional	Not used	-	<core_level_hooks>

• **<groups>**

- **<group name= >** (str) – Lists all groups defined in the core.
 - **<type>** (str) – Group type.
 - **<items>** (str) – List of all items in the group.

<groups>	Data type	Occurrence	Usage	Child tags	Parent tags
<group name>	str	Optional	Not used	-	<group name>

<groups>	Data type	Occurrence	Usage	Child tags	Parent tags
<type>	str	Optional	Not used	-	<group name>
<items>	str	Optional	Not used	-	<group name>

- **<parms>** – List all architectural parameters in the core.
 - **<parm name>**⇒ (str) – Parameter identifier
 - **<value>** (str) – The default value for the parameter.
 - **<options>** (str) – List of valid values for the parameter.

<parms>	Data type	Occurrence	Usage	Child tags	Parent tags
<parm name>	str	Optional	Not used	-	-
<value>	str	Optional	Not used	-	<parm name>
<options>	str	Optional	Not used	-	<parm name>

- **<asm_config>** – List information about the assembler configuration.
 - **<comments>** (str) – List prefixes used to denote the start of a comment.
 - **<line_comments>** (str) – List characters used to denote the start of a single-line comment.
 - **<arch>** (str) – Specifies the architecture used that is given as parameter to the assembler
 - **<attributes>** (str) – Specifies the version for the extensions used
 - **<mattrib>** (str) – Specifies the extensions used by the assembler

<asm_config>	Data type	Occurrence	Usage	Child tags	Parent tags
<comments>	str	Optional	Not used	-	<asm_config>
<line_comments>	str	Optional	Not used	-	<asm_config>
<attributes>	str	Mandatory	Used	-	<asm_config>
<mattrib>	str	Mandatory	Used	-	<asm_config>
<arch>	str	Mandatory	Used	-	<asm_config>

- **<helpers>** – List all helper methods in the core.
 - **<helper name>**⇒ (str) – Helper identifier
 - **<action>** (str) – The code for the helper function.
 - **<inputs>** (str) – Lists source registers or register files.
 - **<helpers>** (str) – List any core-level helper functions used by the helper.
 - **<raises_exceptions>** (str) – If true, the helper may raise an explicit exception.

<helpers>	Data type	Occurrence	Usage	Child tags	Parent tags
<helper name>	str	Optional	Not used	-	-
<action>	str	Optional	Not used	-	<helper name>
<helpers>	str	Optional	Not used	-	<helper name>
<raises_exceptions>	str	Optional	Not used	-	<helper name>
<inputs>	str	Optional	Not used	-	<helper name>

Chapter 8. How to generate register pairs?

Register generation is supported and it is triggered automatically when the tools find specific information in the XML file parsed. In order to activate this feature the user has to make sure that the XML file specifies in **<inputs>** and/or **<outputs>** tags that a certain instruction is using register pairs. For instance, the tools are limited to generate only pairs of 2 registers. An example of how it could look:

```
<outputs>
  <str>GPR(rdc_p)</str>
  <str>GPR(rdc_p + 1)</str>
</outputs>
```

Based on the information, the tools generate definitions for the registers that could be used as pairs. Firstly, the tools generate consecutive pairs of register, indicating informations about the **ABI alias**, the **size** also specifies the internal structure of the register pair (**even - odd**).

```
def X2_X3 : RISCVRegWithSubRegs<2, "X2", [X2, X3], ["sp"]> {
  let SubRegIndices = [sub_gpr_even, sub_gpr_odd];
  let CoveredBySubRegs = 1;
}
def X4_X5 : RISCVRegWithSubRegs<4, "X4", [X4, X5], ["tp"]> {
  let SubRegIndices = [sub_gpr_even, sub_gpr_odd];
  let CoveredBySubRegs = 1;
}
```

After these definitions are generated, the tools will generate the actual register class, which specifies all the register pairs available, based on the calling convention allocation and also based on the register class **width** and **shift** values.

```
// Register Class GPRP : Register Pair
def GPRP : RegisterClass<"RISC", [XLenVT], 64, (
  add X10_X11, X12_X13, X14_X15, X16_X17,
  X6_X7, X28_X29, X30_X31, X8_X9,
  X18_X19, X20_X21, X22_X23, X24_X25,
  X26_X27, X0_X0, X2_X3, X4_X5
)> {
  let RegInfos = XLenRI;
}
```

If the calling convention specifies a special case for **X0 register pair**, the tools check if **X0** is considered **Hard wired zero**, meaning that X0 pair should have a **different definition**. In this case, **DUMMY_REG_PAIR_WITH_X0** will be generated as a special register pair for X0 and then added to the register class X0 belongs to.

```
def DUMMY_REG_PAIR_WITH_X0 : RISCVReg<0, "0">;
def GPRAll : RegisterClass<"RISCV", [XLenVT], 32, (
  add GPR, DUMMY_REG_PAIR_WITH_X0
)> {
  let RegInfos = XLenRI;
}
```

Once register pair generation is based on even-odd structure, these have to be specified.

```
def sub_gpr_even : SubRegIndex<32>;
def sub_gpr_odd : SubRegIndex<32, 32>;
```

The keys and values used in this definitions could be changed or updated in **llvm_config.txt** file.

Chapter 9. How To Generate Intrinsic And Tests?

The tools built are meant to generate intrinsic definitions and test for any ADL model given as input argument. In order to activate this feature, the user should be aware of the information required for proper generation. The tools are able to generate instructions patterns, intrinsic definitions, a header containing the mapping between the LLVM required names for intrinsic definitions and user custom name given to the same intrinsic definitions, but also a list of tests, each test being ready to use.

In order to use all these features, the user has to provide several information in the ADL xml model as it follows:

<intrinsic> (str)

This tag specifies the identifier used for pattern generation. The tools takes this identifier and used it in a pattern definition associated with instruction for which the <intrinsic> tag is defined.

```
def : Pat<(i32 (*int_riscv_add* GPR:$rs1, GPR:$rs2)), (ADD $rs1, $rs2)>;
```

<intrinsic_args> (str)

This tag specifies the intrinsic arguments that is used for generation. Generally, the declaration of a register argument is similar to the <inputs>/<outputs> declaration.

```
<intrinsic_args>
  <str>GPR(rd)</str>
  <str>GPR(rs1)</str>
  <str>GPR(rs2)</str>
</intrinsic_args>
```

<intrinsic_type>

<instrfield_intrinsic name= > (str)

This tag takes each argument previously defined and specifies a data type for this argument. This information is used when defining the intrinsic in a separate file.

```
<intrinsic_type>
  <instrfield_intrinsic name="GPR(rd)">
    <str>llvm_i32_ty</str>
  </instrfield_intrinsic>
</intrinsic_type>
```

```
def int_riscv_add : Intrinsic<[llvm_i32_ty], [llvm_i32_ty,
```

```
llvm_i32_ty], [IntrNoMem]>, ClangBuiltin<"__builtin_riscv_add">;
```

<generate_builtin> (str)

This tag specifies information about the builtin generated for a certain instruction.

```
<generate_builtin>
  <str>__rv_add</str>
</generate_builtin>
```

This identifier is then used in several generated files as it follows:

riscv_builtinRv32i.h

```
#define __rv_add(a, b) __builtin_riscv_add((a), (b))
```

BuiltinRISCVRv32i.def

```
TARGET_BUILTIN(__builtin_riscv_add*, "UiUiUi*", "nc", "rv32i")
```

The files generated which contain all the details about intrinsic and builtin definitions are:

- **BuiltinRISCV<extension>.def**
- **riscv_builtin<extension>.h**
- **RISCVIntrinsics_gen<extension>.td**

An example of an intrinsic defined for ADD instruction on RV32I model

```
<intrinsic>
  <str>int_riscv_add</str>
</intrinsic>
<intrinsic_args>
  <str>GPR(rd)</str>
  <str>GPR(rs1)</str>
  <str>GPR(rs2)</str>
</intrinsic_args>
<intrinsic_type>
  <instrfield_intrinsic name="GPR(rd)">
    <str>llvm_i32_ty</str>
  </instrfield_intrinsic>
</intrinsic_type>
<generate_builtin>
  <str>__rv_add</str>
</generate_builtin>
```

Naming convention is also handled by these tools. In order to ease the usage of builtin defined, the user can give to the builtin an identifier different from the standard required by LLVM. The tools handle this situation by generating a header file in which this naming convention is treated, basically mapping the custom builtin to a proper LLVM builtin definitions. Moreover, in any test or usage of this builtin, the user can call the builtin using the custom name instead of the required by LLVM name. The definition in the header file looks like:

riscv_builtinRv32i.h

```
#define __rv_add(a, b) __builtin_riscv_add((a), (b))
```

Tests Generation

For verifying and validating the builtin definitions, a test is created for each builtin defined. The structure of the test includes the header for naming convention and a function which uses the builtin definition in order to pass the validation. The tests are automatically generated in a customized folder which is generally included in tools/testing/intrinsics/Tests. For a better overview, we take an example:

```
// RUN: %clang --target=riscv32 -march=rv32i %s -S -o %s.s
// RUN: cat %s.s | %filecheck %s
void do_rv_add(int *values_set1, int *values_set2, int *results_rv_add)
{
    *results_rv_add = __rv_add(*values_set1, *values_set2);
}
// CHECK: add a\{\{[0-9]\}\}, a\{\{[0-9]\}\}, a\{\{[0-9]\}\}
```

Chapter 10. How to generate Sail description?

Sail is a language for describing the instruction-set architecture (ISA) semantics of processors: the architectural specification of the behaviour of machine instructions.

NOTE For more information about **Sail**: <https://github.com/riscv/sail-riscv>

Sail description generation is a feature supported in ADL tools for having, beside TD files generation, the **RISC-V** extension formal specification written in Sail for instruction encoding, semantic and parsing/decoding information.

• How it works?

- In order to enable Sail description generation, the user should run the basic **make-td** command line. When this command is run, a **.sail** file is generated for each RISC-V extension which is specified.
- When the Sail description generation is activated, the previous generated Sail file is automatically deleted and replaced by the fresh generated files.

NOTE

For SAIL the user has to set the extensions using the **--extension** flag. The order of the attributes from the command line matters. If an instruction has more than one attribute it will only be generated once for the first attribute provided at **--extension** flag (this may result in empty files being generated).

• How a Sail description looks?

- Naming convention for Sail description files could be set in **llvm-config** by editing **SailDescription** field.
- Each instruction supported contains 4 parts defined: union clause ast, mapping clause encdec, function clause execute, mapping clause assembly.
- For defining an extension, there is **enum clause extension definition** where the extension enabled is defined.

```
enum clause extension = Ext_Zclsd
```

- Beside extension definition, other dependencies are defined (e.g. other extensions which are required). Any extension which represents a dependency is then generated with **extensionEnabled()**, except from **rv32i** extension which is covered by **xlen == 32** definition.
- **sys_enable()** definition is also required for the extension defined, except from **rv32i** extension.

```
function clause extensionEnabled(Ext_Zclsd) = xlen == 32 & sys_enable_zclsd() &  
extensionEnabled(Ext_Zca) & extensionEnabled(Ext_Zilsd)
```

- Union clause ast is a formal description for the instruction mnemonic in which it also specifies which type of instruction fields is used (registers or immediates).

```
union clause ast = ADD : (regidx, regidx, regidx)
```

- Mapping clause encdec provides information about the encoding of the instruction. It also specifies the exact range of bits provided for an instruction field, immediate value or opcode.

```
mapping clause encdec = ADD(rs2, rs1, rd)
  if xlen == 32
  <-> 0b00000000 @ rs2 : bits(5) @ rs1 : bits(5) @ 0b000 @ rd : bits(5) @ 0b0110011
  if xlen == 32
```

- Function clause execute is the main part of the Sail description in which information about how the instruction behaves when executed. It also provides information about how the registers work, how the information is parsed, how the memory is handled or which operands and operators are used. Moreover, there are several exceptions which are defined and handled.

```
function clause execute(ADD(rs2, rs1, rd)) = {
  let rs2_val = X(rs2);
  let rs1_val = X(rs1);
  let rd_val = X(rd);
  let result : xlenbits = rs1_val + rs2_val in
  X(rd) = result;
  RETIRE_SUCCESS
}
```

- Mapping clause assembly translates the instruction in Sail language preserving information about registers or instruction fields.

```
mapping clause assembly = ADD(rs2, rs1, rd)
  if xlen == 32
  <-> "add" ^ spc() ^ reg_name(rs2) ^ sep() ^ reg_name(rs1) ^ sep() ^ reg_name(rd)
  if xlen == 32
```

NOTE

For other extension than rv32i, extensionEnabled() definitions will be generated for activating the correct extension:

```
mapping clause assembly = ZCLSD_C_LDSP(imm, rx)
  if extensionEnabled(Ext_Zca) & extensionEnabled(Ext_Zclsd) &
  extensionEnabled(Ext_Zilsd) & xlen == 32
  <-> "c.ldsp" ^ spc() ^ reg_name(rx) ^ sep() ^ hex_bits_9(imm @ 0b000)
```



```
if extensionEnabled(Ext_Zca) & extensionEnabled(Ext_Zclsd) &  
extensionEnabled(Ext_Zilsd) & xlen == 32
```

- **Which type of instructions are supported?**

- For instance, the instructions supported for Sail description generation are: **R-type** instructions, **I-type** instructions (**Load** instructions included) and **S-type** instructions. Although, for a proper generation, the action specified inside **<action>** tag in the XML file is vital. It should be double-checked and verified for not having generation issues. Also, the script uses other fields from the XML file such as **<syntax>** tag, **<inputs>**, **<outputs>** and **<fields>**, so it would be recommended to check running the tool.

Chapter 11. How to generate scheduling description?

ADL tools project can generate scheduling description for several extensions. This tool generates a scheduling table description, which contains more information about the core, but also provide scheduling information for each instruction from the extension supported.

- **How it works?**

- Scheduling description is generated automatically when the tool is run and the information used for generating is parsed from the XML file.
- The XML contains `<sched-table>` tag which provide a high-level description for a scheduling model. It specifies the model type, functional units, instruction types, instruction latency and throughput and other scheduling options.

NOTE

All the fields listed below map on **LLVM MCSchedModel** structure from **MCSchedule.h** file. Generally, **each field** from this **scheduling model** suggests which field from **MCSchedModel** structure is actually set.

- **Scheduling model name**

```
<sched-table name="zilsd">
```

- **MicroOpBufferSize** specifies if the model is in order or out of order based on the value given. This field maps `llvm::MCSchedModel::MicroOpBufferSize`

```
<MicroOpBufferSize>
<int>0</int>
</MicroOpBufferSize>
```

- **IssueWidth** specifies if the model is single-issue, dual-issue or other type. This field maps `llvm::MCSchedModel::IssueWidth`

```
<IssueWidth>
<int>1</int>
</IssueWidth>
```

- **MispredictPenalty** specifies the penalty in case of miss for change-of-flow/branch instructions. This field maps `llvm::MCSchedModel::MispredictPenalty`

```
<MispredictPenalty>
<int>5</int>
</MispredictPenalty>
```

- ***LoopOpBufferSize*** specifies the number of micro operations that the processor may buffer for optimized loop execution. This field maps `llvm::MCSchedModel::LoopMicroOpBufferSize`

```
<LoopOpBufferSize>
<int></int>
</LoopOpBufferSize>
```

- ***HighLatency*** specifies the highest expected latency for certain operations which generally have very high latency. This field maps `llvm::MCSchedModel::HighLatency`

```
<HighLatency>
<int></int>
</HighLatency>
```

NOTE

LoadLatency parameter (This field maps `llvm::MCSchedModel::LoadLatency`) is set automatically based on the **maximum latency** obtained from **Load** instructions.

NOTE

CompleteModel (This field maps `llvm::MCSchedModel::LoadLatency`) and **UnsupportedFeatures**(defined in `SchedMachineModel` class from `llvm/Target/TargetSchedule.td`) have to be set manually by the user.

- ***functional-unit*** specifies the functional units used for the scheduling model

```
<functional-unit name="RISCVPipelineMEMORY">
```

- ***doc*** specifies a brief description of the functional unit

```
<doc>
<str>Load/Store/Memory operations</str>
</doc>
```

- ***BufferSize*** specifies the number of resources that may be buffered. This field maps `llvm::MCProcResourceDesc::BufferSize`

```
<BufferSize>
<int>0</int>
</BufferSize>
```

- ***proc_resource*** specifies the number of resources of this kind. This field maps `llvm::MCProcResourceDesc::NumUnits`

```
<proc_resource>
```

```
<int>1</int>
</proc_resource>
```

- ***instruction-sched*** contains instructions types, latency, throughput and functional units used

```
<instruction-sched name="ST">
```

- ***instruction_list*** specifies for which instructions these options are applied

```
<instruction_list>
<str>c.sd,c.sdsp,sd</str>
</instruction_list>
```

- ***latency*** specifies the latency used for the instructions listed above. This field maps ***llvm::MCWriteLatencyEntry::Cycles**

```
<latency>
<int>2</int>
</latency>
```

- ***single_issue*** allows the processor to mark some scheduling classes as single-issue. It maps **SingleIssue** parameter from **ProcWriteResources** class

NOTE

More information about this tag could be find in **llvm/include/llvm/Target/TargetSchedule.td** file.

```
<single_issue>
<int>1</int>
</single_issue>
```

- ***num_micro_ops*** specifies if a subtarget requires multiple micro-ops to write a single result. By default this field is set to 1. It maps **NumMicroOps** parameter from **ProcWriteResources** class

NOTE

More information about this tag could be find in **llvm/include/llvm/Target/TargetSchedule.td** file.

```
<num_micro_ops>
<int>3</int>
</num_micro_ops>
```

- ***pipelines*** specifies which pipeline is used for the instruction list specified previously

```
<pipelines>
<pipeline name="RISCVPipelineMEMORY">
```

- ***acquire_at_cycles*** specifies the cycle at which the resource is acquired by an instruction. It maps **AcquireAtCycles** parameter from **ProcWriteResources** class

NOTE

More information about this tag could be find in **llvm/include/llvm/Target/TargetSchedule.td** file.

```
<acquire_at_cycles>
<int></int>
</acquire_at_cycles>
```

- ***throughput*** specifies the cycle at which the resource is released by an instruction. It maps **ReleaseAtCycles** parameter from **ProcWriteResources** class

NOTE

More information about this tag could be find in **llvm/include/llvm/Target/TargetSchedule.td** file.

```
<throughput>
<int>2</int>
</throughput>
```

- **These tags below provide informations about forwarding such as the resources for which it applies and the value used for ReadAdvance**

```
<forwarding>
```

- ***read_resource*** allows the user to specify only a read resource or create a class by grouping more resources

```
<read_resource name="">
```

- ***value*** specifies the value for ReadAdvance forwarding

```
<value>
<int></int>
</value>
```

- ***id*** specifies the id of the resource in case of multiple Read resources

```
<id>
```

```
<int></int>
</id>
```

- In case of a resource group, this tag allows the user to specify the resources that have to be grouped

```
<resource_list>
<str></str>
</resource_list>
```

NOTE

Scheduling description is entirely based on the information parsed from **<sched-table>** tag and it should be correctly and completely defined for a proper generation.

• How scheduling description looks?

- Generating scheduling is a complex process which provides a scheduling table description (containing core information), a resource description (in which read and write resources are defined) and most important a scheduling description file.
- Scheduling description is the file in which each type of instructions has scheduling information added, such as: functional units used, latency, throughput or other scheduling details.
- Naming convention for these scheduling file could be set in **llvm-config** by editing **ScheduleFile**, **SchedulePath** and **ScheduleFileTable** variables, providing information about the path and the file's name.

```
def ZILSDModel : SchedMachineModel {
```

- Information based on **<MicroOpBufferSize>** tag from **<sched-table>** for defining if it is in-order (=0) or out-of-order (>1)

```
let MicroOpBufferSize = 0;
```

- Information parsed from **<IssueWidth>** tag which specifies if dual-issue is activated (=2), single-issue (=1) or other types (>2) are enabled

```
let IssueWidth = 1;
```

- Information based on **<LoadLatency>** tag which pre-defines latency for load instructions

```
let LoadLatency = 2;
```

- Information parsed from **<MispredictPenalty>** tag for change-of-flow/branch instruction

penalty

```
let MispredictPenalty = 5;
```

```
}
```

```
let SchedModel = ZILSDModel in {
```

- Explicitly set to zero since this core is **in-order**

```
let BufferSize = 0 in {
```

- Functional units are specified
- **ProResource** defines the processor's resources for defining scheduling

```
def RISCVPipelineMEMORY : ProcResource<1>;  
}
```

- **WriteRes** defines new subtarget SchedWriteRes that maps resources the for a target
- It specifies which resources are required, duration, pipeline

```
def : WriteRes<WriteST, [RISCVPipelineMEMORY]> {  
  let Latency = 2;  
  let ResourceCycles = 2;  
}  
def : WriteRes<WriteLD, [RISCVPipelineMEMORY]> {  
  let Latency = 2;  
  let ResourceCycles = 2;  
}
```

- Defines new subtarget **SchedReadAdvance** that maps information for a target **SchedRead**
- Used to model forwarding and considered an advanced modeling feature

```
def : ReadAdvance<ReadST, 0>;  
def : ReadAdvance<ReadLD, 0>;  
}
```

- **Scheduling tests integrating llvm-mca and llvm-lit**

- Once the scheduling description model is done, it should be tested and validated for avoiding possible issues.

- It is very important to check an execution timeline when implementing a new scheduling model.
- The latency and throughput defined for each instruction together with the functional units used are key elements for a correct timeline execution.
- That is the reason behind generating scheduling tests. ADL tools project provide scheduling tests for each instruction parsed from the XML file.
- Moreover, references are also generated for comparing and validating results. The mechanism consists in:
- **Generating instructions tests**

```
ld s8, 2(t2)
ld s0, 2(t0)
```

- **Generating reference based on the timeline produced by llvm-mca**

```
// CHECK:      [0,0] DeE .  ld s8, 2(t2)
// CHECK-NEXT: [0,1] . DeE  ld s0, 2(t0)
```

NOTE

CHECK and **CHECK-NEXT** are commands for llvm-lit which specify the execution order

- **Integrate llvm-mca and llvm-lit commands**

```
// RUN: %llvm-mca -mtriple=riscv32 -mcpu=core-name -timeline --timeline-max
-cycles=0 -iterations=1 %s &> %s.txt
// RUN: cat %s.txt | %filecheck %s
```

- Basically, **llvm-mca** runs the instruction and generate an execution timeline which is compared to the reference using **llvm-lit**.
- After running the tool, 2 types of scheduling tests are generated: basic tests and data-dependency tests.
- Data-dependency tests use the destination of the first instruction as source in the second instruction tested, if applicable. On the other hand, the basic tests use totally different registers, randomly chosen.
- For validating the results and checking the scheduling model, llvm-lit allows to run an entire test suite and then check if tests passed or failed.
- Beside **llvm-lit**, **llvm-mca** should be activated in the command line, together with **File Check**.

```
<path>/llvm-lit --param app_filecheck=<path>/llvm-build/bin/FileCheck --param
app_llvm_mca=<path>/llvm-build/bin/llvm-mca <tests_folder>
```


NOTE

Core-name is generated automatically based on the information parsed from **<sched-table name= "">** tag.

NOTE

This command line shows the result from every single test from the test suite run and also provide a **.txt** file which serves as log, for more information about certain tests.

Chapter 12. Known Limitations

- **Fix-up relocations not supported**
- **Scheduling information is for demo purposes**