CODASIP URISC INSTRUCTION SET REFERENCE MANUAL

Document Version	1.0
Core Version	5.0.0
Release Date	March 04, 2020

COPYRIGHT AND PROPRIETARY NOTICE

Codasip uRISC Instruction Set Reference Manual

INTEGRATED 3RD PARTY SOFTWARE MODULES AND THEIR LICENSES

In addition to the Codasip License Agreement, integrated 3rd party software is also licensed for use under the terms and conditions of the license agreements set forth in the document titled "INTEGRATED 3RD PARTY SOFTWARE MODULES AND THEIR LICENSES" which is located on the Codasip Website.

TABLE OF CONTENTS

1 PREFACE

1.1 About

This document describes the overall architecture of the Codasip™ uRISC processor, its instruction set and instruction format of each instruction. Specification of syntax, semantics, description of functionality, binary encoding and examples are provided.

1.1.1 Intended Audience

This document is intended for developers using Codasip Studio and working with Codasip uRISC processor. This document can be used for developing applications for Codasip uRISC processor architecture without any prior knowledge of CodAL™ language.

1.1.2 Release Information

1.0.0 — Initial version

1.1.3 Product Revisions

Codasip uRISC 5.0.0

This reference guide can be used with any version of Codasip Studio.

1.1.4 Typographical Conventions

Tab. 1: Typographical conventions

Convention	Usage	Example
Capitalized	Standardized terms, defined earlier in the text or in the Glossary.	Window, Project
Italics	Important text, term or additional information.	*Do not forget to ...*
Bold	Inline references to keywords of the IDE (usually starts in upper case).	The **Project Explorer** window
`Monospace`	Code, code values, Unix file names, prompts or instructions.	File name `ca_utils.hcodal`
Document ref	Reference to Codasip and other documents.	Please refer to the *CodAL Language Reference Manual*.
<abstract name>	Field for substitution with user data.	*<model>*
`keyword`	Inline references to CodAL™ keywords (lower case).	`element`, `event`
Option→Suboption	Command path, typically starting from the main	**File → New → CodAL Project**

Convention	Usage	Example
	toolbar.	
`Example`	Examples - typically snippets of code.	`register bit[DATA_W] test;`
`Syntax`	Explanation of syntax.	`StartSection: "start" "{"`
WARNING:	Warning against unexpected behavior.	**WARNING:** Changing this value can cause error.
NOTE:	Information useful for the user.	**NOTE:** When in troubles, contact Support.
MESSAGE:	Messages that can be shown in Codasip Studio.	**MESSAGE:** Invalid value.

1.2 References

1.2.1 Other Codasip Documents

Here is a complete list of the documentation for Codasip Studio:

Guides

Document	Description
Codasip License Setup and Installation Guide	Detailed description of licenses setup.
Codasip Studio Installation Guide	How to install the Codasip Studio software package.
Codasip Studio User Guide	Detailed guidance on the use of Codasip Studio and the tools that it contains.
Codasip Studio SDK User Guide	A complete reference and guide to usage of SDKs generated by Codasip Studio.

Reference Manuals

Document	Description
CodAL Language Reference Manual	A complete presentation of the CodAL language and how to use it for writing processor models.
Codasip Studio Message Reference Manual	A list of Codasip errors, warnings and notes that user can encounter during his work with Codasip Studio with descriptions, explanations, and possible solutions.
Codasip Program Description Model Language Manual	A complete presentation of the PDML language and how to use it for writing constraints for random applications generator.
Codasip Studio Technical Reference Manual	Reference information on Codasip Studio and the tools that it contains.

Tutorials

Document	Description
Codasip Studio Quick Start Tutorial	A step-by-step introduction to the essentials of Codasip Studio.
Codasip Instruction Accurate Model Tutorial	A step-by-step introduction to writing Instruction Accurate processor models in CodAL.
Codasip Compiler Generation Tutorial	A step-by-step introduction to generating a C/C++ compiler from an Instruction Accurate processor model written in CodAL.
Codasip Cycle Accurate Model Tutorial	A step-by-step introduction to writing a Cycle Accurate CodAL model.

Document	Description
Codasip Interrupts and Peripherals Tutorial	A step-by-step introduction to adding external devices to an processor CodAL model.
Codasip JTAG Extension Tutorial	A step-by-step introduction to Codasip's JTAG extension.
Codasip SIMD Extension Tutorial	Tutorial showing the implementation of SIMD extensions in the Codasip uRISC.
Codasip Custom Components Verification Tutorial	Tutorial describing process of adding manually modified UVM test-bench for a component into the processor UVM test-bench.
Codasip uRISC VLIW Extension Tutorial	Tutorial showing modifications to Codasip uRISC to create a simple VLIW architecture.

1.2.2 Other References

None.

1.3 Feedback

1.3.1 Feedback on Codasip Products

If you have any comments or suggestions about Codasip products, please contact your supplier or send an email to support@codasip.com. Please provide:

- The product name
- The product version
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

1.3.2 Feedback on this Document

If you have comments on this document, please send an email to feedback@codasip.com. Please provide:

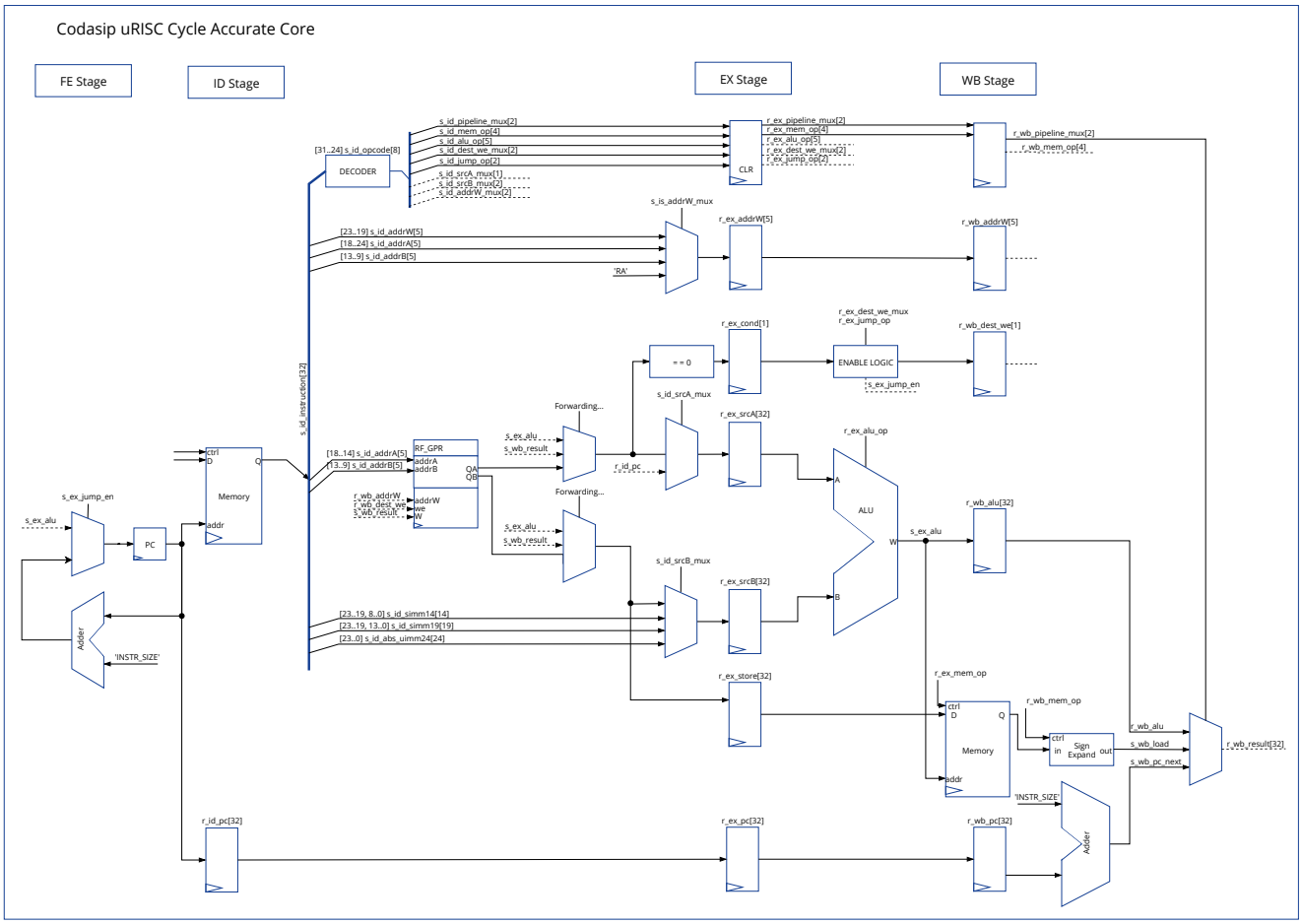
- The document title and format (pdf, web page, etc)
- The document version
- The chapter number and page numbers to which your comments apply
- A concise explanation of your comments.

Codasip also welcomes general suggestions for additions and improvements.

2 INTRODUCTION

Codasip™ uRISC microprocessor is a 32-bit general-purpose processor which implements a simple microarchitecture. Its main intent is to provide basic guidelines for describing processor architecture in CodAL™ language and generating toolchain and applications using Codasip Studio. Having simple instruction set and underneath microarchitecture design, it is suited for learning and tutorial purposes in obtaining the necessary knowledge for designing more complex processor architectures.

Fig. 1: Codasip uRISC microarchitecture



2.1 Architectural Resources

2.1.1 Program Counter

Program counter is the only special purpose register of Codasip uRISC processor and it is 32-bit wide. Value of the program counter is used for addressing the program memory and fetching the instructions. Addresses are

always aligned to 4 bytes.

2.1.2 General Purpose Registers

Codasip uRISC processor contains general purpose register file with 32 registers. Each register is 32-bit wide. In assembly syntax they are denoted as R0-R31.

All registers are accessible for programming, the next table summarizes the function and description of registers.

Register	Name	Type	Reset value	Description
R0	-	RW	0	Stack pointer
R1	-	RW	0	Base pointer
R2	-	RW	0	Auxiliary register
R3	-	RW	0	Return address
R4	-	RW	0	Arguments/ Return value
R5	-	RW	0	Arguments/ Return value
R6	-	RW	0	Arguments/ Return value
R7	-	RW	0	Arguments/ Return value
R8	-	RW	0	
R9	-	RW	0	
R10	-	RW	0	
R11	-	RW	0	
R12	-	RW	0	
R13	-	RW	0	
R14	-	RW	0	
R15	-	RW	0	
R16	-	RW	0	
R17	-	RW	0	
R18	-	RW	0	
R19	-	RW	0	
R20	-	RW	0	
R21	-	RW	0	
R22	-	RW	0	
R23	-	RW	0	
R24	-	RW	0	
R25	-	RW	0	
R26	-	RW	0	
R27	-	RW	0	
R28	-	RW	0	
R29	-	RW	0	
R30	-	RW	0	
R31	-	RW	0	

2.1.3 Program and data memories

Memory for storing instructions and data has 32-bit address interface and memory is organized to directly address 32-bit words with least addressable unit of 8-bit word. Codasip uRISC has Von Neumann architecture with unified address space for code and data. All memory accesses are managed as big-endian.

2.1.4 Interfaces

Codasip uRISC has two interfaces to connect memory with code and data. Both of the memories are of type MEMORY:MASTER. Both of them can read 32-bit wide words, only the data interface allows to read and write data from and to the memory.

3 DATA LAYOUT

Data layout is the same as in other standard 32-bit processors; standard C language types have the following sizes and alignment:

Tab. 2: C language data sizes and alignment.

Type	Size (bytes)	Alignment (bytes)
`long long`	8	4
`long`	4	4
`int`	4	4
`short`	2	2
`char`	1	1
`pointers`	4	4

3.1 Addressing modes

The uRISC processor support the following addressing modes for data:

- **register** – the operand is stored in the given register; see sections "Conditional Move Instructions" and "Arithmetic, Logic and Comparison Instructions".
- **immediate** – an immediate operand is encoded directly in the instruction word; see section "Move Instructions with Immediate Operand".
- **displacement** – the destination address to memory is calculated as a sum of base address in given register and signed immediate operand; see section "Load Instructions" and "Store Instructions ".

4 INSTRUCTION TERMINOLOGY

This section describes the instruction terminology used throughout this document.

Abbreviation	Name	Size	Description
OPC	Operation code	8 bits	Operation code of instruction.
GPR	General Purpose Register	32 bits	General purpose register from the register file. Each GPR is 32 bits wide.
IMMX	Immediate value	X bits	Immediate value encoded at X bits
UNUSED	Unused bits	variable	Unused bits in instruction word padded with zeros.
PC	Program counter	32 bits	Program counter carrying address of next instruction to be fetched from the program memory.

Possible references to general purpose registers can be as follows:

General Purpose Register	Description
SRC_i	GPR with specific index used as source operand.
DST	GPR used as destination operand.
COND	GPR used for condition evaluation.

Possible immediate values are as follows with bitwidth as specified for particular instruction:

Immediate value	Description
SIMM	Signed immediate operand used in arithmetic and move operations.
ABS_ADDR	Absolute destination address of next instruction.
REL_ADDR	Relative address to current value of program counter.

5 ASSEMBLY LANGUAGE SYNTAX

5.1 Assembly Language Comments

Supported format of comments in assembly language for this core by Codasip SDK:

Ex. 1: Assembly Language comments

```
// line comment
/* block comment */
```

5.2 Jump Label

Supported jump label for this core by Codasip SDK:

Ex. 2: Jump label

```
label_name:
...
// jump instruction on previously defined jump label
jmp label_name
```

5.3 Define statement

Following is the syntax of define statement in assembler for this core:

Ex. 3: Define statement directive

```
.equiv define_name, define_value
```

Note that `define_name` is case sensitive. `define_value` should be number in decimal (without any prefix), hexadecimal (prefix `0x`) or binary (prefix `0b`) format. Codasip tools can also handle basic preprocessor computation.

5.4 Directives

All assembler directives have names that begin with a period ('.'). The names are case insensitive and usually written in lower case.

Example of directive, that creates code section:

Ex. 4: .text directive

```
.text
addi R1, 2
nop
...
```

Codasip directives are inspired by GNU assembly language directives, most of them are supported as well, see website GNU assembler for more information. For complete list of supported directives by Codasip Assembler follow *Codasip Studio Technical Reference Manual*.

5.5 Inline Assembly Format

Format of the Codasip Inline Assembly is based on GNU Inline Assembly Format. It can be basic or extended. The extended format can use C expression operands while the basic cannot.

Syntax of basic format follows.

```
(asm|__asm__) [volatile] ("AssemblyCode");
```

Specifier `volatile` has no effect, because all basic inline assembly are `volatile` by default.

Ex. 5: Basic inline assembly format

```
asm ("nop");
```

The basic format can be used outside of functions and it is fully controlled by the user. The compiler does not understand specified assembler code. For additional information about semantics of basic format, see Basic-Asm.html.

Syntax of extended format follows:

```
(asm|__asm__) [volatile] ("AssemblyCode" : OutputOperands [ : InputOperands [ :
Clobbers]]);
```

`AssemblyCode` is a string constructed from a text and references to operands, the operands are denoted by `%`.

`OutputOperands`, `InputOperands` and `Clobbers` are ordered lists of items and are delimited with a comma.

Syntax of output operand follows:

```
[SymbolicName] "(=|+)(r|m)" (Variable)
```

Syntax of input operand follows:

```
[SymbolicName] "(r|m|i|Constant)" (Variable|Expression)
```

Clobbers are registers that are modified as a side-effect by inline assembly; it is a list of register names.

Ex. 6: Extended inline assembly format

```
int foo(int a, int b)
{
    int c;
    __asm__("add %0, %[first], %2"
        : "=r" (c)
        : [first]"r"(a), "r"(b)
        : "rc");
    return c;
}
```

The compiler partially understands the specified assembler code. For additional information about semantics of extented format, see Extended-Asm.html.

6 NOTE ON CODASIP SDK

The generated toolchain from Codasip Studio covers complete set of tools for development, testing and debugging. Codasip SDK contains also instruction-accurate and cycle-accurate simulator which allow to simulate programs written for specific microprocessor and behaviour of components acting as its peripherals.

The instruction accurate simulator contains support for calling system functions which allow easier debugging with using e.g. `printf` functions (this support is provided through special `syscall` instruction in the instruction-accurate model of the processor). The simulator and the tools are generated with the support of the *newlib* library. Therefore, to quickly verify the functionality of the application (or complete system) it is possible to directly use e.g. `printf()` [1] functions and the tools handle the rest automatically.

For the usage of the Codasip SDK, please refer to the *Codasip Studio SDK User Guide*.

For setting up license server, please refer to the Codasip documentation described below:

- *Codasip Studio Installation Guide*, chapter "Downloading & Installation"

[1] Note that this is possible only for instruction-accurate simulator. Cycle-accurate simulator does not support these special functions because they are relevant only for the software level.

7 INSTRUCTION SET

The instruction set consists of 32-bit instructions. There are several instructions formats depending on the particular group in instruction set. The following sections describe each instruction format in more detail.

7.1 Special Instructions

There are two special instructions in the Codasip uRISC instruction set. Their instruction format is depicted in the following table:

31 24	23 0
OPC	UNUSED
8	24

Field descriptions:

Field	Description
OPC	Operation code of instruction.
UNUSED	Unused bits, filled with zeros.

7.1.1 NOP

Instruction	**NOP**
Op. code	0x00
Syntax	nop
Semantics	-
Latency	1

31 24	23 0
0x00	0x0
8	24

Example	nop
Description	No operation. Used for inserting wait cycles.

7.1.2 HALT

Instruction	**HALT**

Op. code	0x01
Syntax	halt
Semantics	simulation_stop()
Latency	1

31 24	23 0
0x01	0x0
8	24

Example	halt
Description	This instruction stops the simulation.

7.2 Conditional Move Instructions

7.2.1 MOVZ

Instruction	**MOVZ**
Op. code	0x20
Syntax	DST = movz SRC_1, SRC_2
Semantics	if (SRC_1 == 0) DST ← SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x20	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = movz r2, r3
Description	Copies value of source SRC_2 to destination DST register, if SRC_1 equals zero.

7.2.2 MOVNZ

Instruction	**MOVNZ**
Op. code	0x21
Syntax	DST = movnz SRC_1, SRC_2
Semantics	if (SRC_1 != 0) DST ← SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x21	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = movnz r2, r3
Description	Copies value of source SRC_2 to destination DST register, if SRC_1 does not equal zero.

7.3 Move Instructions with Immediate Operand

Instructions working with immediate operand (constant value) allow to encode 19b value into the instruction word. In this group, instructions for loading and moving immediate values are described.

31 24	23 19	18 14	13 0
OPC	SIMM	DST	SIMM
8	5	5	14

Field descriptions:

Field	Description
OPC	Operation code of instruction.
DST	Destination GPR.
SIMM	Signed immediate.

7.3.1 MOVSI

Instruction	**MOVSI**
Op. code	0x02
Syntax	DST = movsi SIMM
Semantics	DST ← SIMM
Latency	1

31 24	23 19	18 14	13 0
0x02	SIMM	DST	SIMM
8	5	5	14

Example	r1 = movsi 2
Description	Moves 19-bit signed immediate operand to DST register.

7.3.2 MOVHI

Instruction	**MOVHI**
Op. code	0x03
Syntax	DST = movhi SIMM
Semantics	DST ← SIMM[15..0] :: DST[15..0]
Latency	1

31 24	23 19	18 14	13 0
0x03	SIMM	DST	SIMM
8	5	5	14

Example	r1 = movhi 2
Description	Moves 16-bit signed immediate operand to top DST register.

7.4 Arithmetic, Logic and Comparison Instructions

There are several instructions performing typical arithmetic, logic and comparison instructions described in the following subsections.

31 24	23 19	18 14	13 9	8 0
OPC	DST	SRC_1	SRC_2	UNUSED
8	5	5	5	9

Field descriptions:

Field	Description
OPC	Operation code of instruction.
DST	Destination GPR.
SRC_1	First source GPR.
SRC_2	Second source GPR.
UNUSED	Unused bits, filled with zeros.

7.4.1 ADD

Instruction	**ADD**
Op. code	0x05
Syntax	DST = add SRC_1, SRC_2
Semantics	DST ← SRC_1 + SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x05	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = add r2, r3
Description	Performs addition of values in SRC_1 and SRC_2 GPR and stores the result in the DST GPR.

7.4.2 SUB

Instruction	**SUB**
Op. code	0x06
Syntax	DST = sub SRC_1, SRC_2
Semantics	DST ← SRC_1 - SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x06	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = sub r2, r3
Description	Performs subtraction of values in SRC_1 and SRC_2 GPR and stores the result in the DST GPR.

7.4.3 MUL

Instruction	**MUL**
Op. code	0x07
Syntax	DST = mul SRC_1, SRC_2
Semantics	MUL ← SRC_1 * SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x07	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = mul r2, r3
Description	Performs multiplication of values in SRC_1 and SRC_2 GPR and stores the result in the DST GPR. The result is truncated to 32 bits.

7.4.4 AND

Instruction	**AND**
Op. code	0x08
Syntax	DST = and SRC_1, SRC_2
Semantics	DST ← SRC_1 & SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x08	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = and r2, r3
Description	Performs logical AND of values in SRC_1 and SRC_2 GPR and stores the result in the DST GPR.

7.4.5 OR

Instruction	**OR**
Op. code	0x09
Syntax	DST = or SRC_1, SRC_2
Semantics	DST ← SRC_1 | SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x9	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example r1 = or r2, r3

Description Performs logical OR of values in SRC_1 and SRC_2 GPR and stores the result in the DST GPR.

7.4.6 XOR

Instruction **XOR**

Op. code 0x0A

Syntax DST= xor SRC_1, SRC_2

Semantics DST ← SRC_1 ^ SRC_2

Latency 1

31 24	23 19	18 14	13 9	8 0
0x0A	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example r1 = xor r2, r3

Description Performs logical exclusive-OR of values in SRC_1 and SRC_2 GPR and stores the result in the DST GPR.

7.4.7 SLL

Instruction **SLL**

Op. code 0x0B

Syntax DST= sll SRC_1, SRC_2

Semantics DST ← SRC_1 << SRC_2[4..0]

Latency 1

31 24	23 19	18 14	13 9	8 0
0x0B	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example r1 = sll r2, r3

Description Performs logical shift left operation of value in SRC_1, shift length is stored in SRC_2 GPR. Result is stored in the DST GPR.

7.4.8 SRL

Instruction	**SRL**
Op. code	0x0C
Syntax	DST = srl SRC_1, SRC_2
Semantics	DST ← SRC_1 (u) >> SRC_2[4..0]
Latency	1

31 24	23 19	18 14	13 9	8 0
0x0C	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = srl r2, r3
Description	Performs logical shift right operation of value in SRC_1 with no sign extension, shift length is stored in SRC_2 GPR. Result is stored in the DST GPR.

7.4.9 SRA

Instruction	**SRA**
Op. code	0x0D
Syntax	DST = sra SRC_1, SRC_2
Semantics	DST ← SRC_1 (s) >> SRC_2[4..0]
Latency	1

31 24	23 19	18 14	13 9	8 0
0x0D	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = sra r2, r3
Description	Performs arithmetic shift right operation of value in SRC_1 with sign extension, shift length is stored in SRC_2 GPR. Result is stored in the DST GPR.

7.4.10 EQ

Instruction	**EQ**
Op. code	0x1A
Syntax	DST = eq SRC_1, SRC_2
Semantics	DST ← (SRC_1 == SRC_2)
Latency	1

31 24	23 19	18 14	13 9	8 0
0x1A	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = eq r2, r3
Description	Sets value of DST GPR to non-zero when SRC_1 and SRC_2 GPR are equal, otherwise DST GPR is set to zero value.

7.4.11 NEQ

Instruction	**NEQ**
Op. code	0x1B
Syntax	DST= neq SRC_1,SRC_2
Semantics	DST ← (SRC_1 != SRC_2)
Latency	1

31 24	23 19	18 14	13 9	8 0
0x1B	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = neq r2, r3
Description	Sets value of DST GPR to non-zero when SRC_1 and SRC_2 GPR are not equal, otherwise DST GPR is set to zero value.

7.4.12 SLT

Instruction	**SLT**
Op. code	0x1C
Syntax	DST= slt SRC_1,SRC_2
Semantics	DST ← (SRC_1 (s)< SRC_2)
Latency	1

31 24	23 19	18 14	13 9	8 0
0x1C	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example	r1 = slt r2, r3
Description	Performs signed comparison of values stored in SRC_1 GPR and SRC_2 GPR. If value in SRC_1 GPR is lower than value stored in SRC_2 GPR, DST GPR is set to non-zero value. Otherwise it is set to zero.

7.4.13 ULT

Instruction	**ULT**
Op. code	0x1D
Syntax	DST= ult SRC_1,SRC_2
Semantics	DST ← (SRC_1 (u)< SRC_2)

Latency 1

31 24	23 19	18 14	13 9	8 0
0x1D	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example r1 = ult r2, r3

Description Performs unsigned comparison of values stored in SRC_1 GPR and SRC_2 GPR. If value in SRC_1 GPR is lower than value stored in SRC_2 GPR, DST GPR is set to non-zero value. Otherwise it is set to zero.

7.4.14 SLE

Instruction **SLE**

Op. code 0x1E

Syntax DST = sle SRC_1,SRC_2

Semantics DST ← (SRC_1 (s)≤ SRC_2)

Latency 1

31 24	23 19	18 14	13 9	8 0
0x1E	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example r1 = sle r2, r3

Description Performs signed comparison of values stored in SRC_1 GPR and SRC_2 GPR. If value in SRC_1 GPR is lower than or equal to the value stored in SRC_2 GPR, DST GPR is set to non-zero value. Otherwise it is set to zero.

7.4.15 ULE

Instruction **ULE**

Op. code 0x1F

Syntax DST= ule SRC_1,SRC_2

Semantics DST ← (SRC_1 (u)≤ SRC_2)

Latency 1

31 24	23 19	18 14	13 9	8 0
0x1F	DST	SRC_1	SRC_2	0x0
8	5	5	5	9

Example r1 = ule r2, r3

Description Performs unsigned comparison of values stored in SRC_1 GPR and SRC_2 GPR. If value in SRC_1 GPR is lower than or equal to the value stored in SRC_2 GPR, DST GPR is set to non-zero value. Otherwise it is set to zero.

7.5 Arithmetic and Logic Instructions with Immediate Operand

Instructions in this section perform arithmetic operations with register and immediate operand. Currently, there is only one instruction that performs addition operation.

31 24	23 19	18 14	13 9	8 0
OPC	SIMM	SRC	DST	SIMM
8	5	5	5	9

Field descriptions:

Field	Description
OPC	Operation code of instruction.
DST	Destination GPR.
SRC	Source GPR.
SIMM	14b immediate operand.

7.5.1 ADDI

Instruction	**ADDI**
Op. code	0x24
Syntax	DST = addi SRC, SIMM
Semantics	DST ← SRC + (int32)SIMM
Latency	1

31 24	23 19	18 14	13 9	8 0
0x24	SIMM	SRC	DST	SIMM
8	5	5	5	9

Example	r1 = addi r2, 3
Description	14-bit immediate value is sign extended to 32 bits and the result is added with the content of register SRC GPR. Result is written into the DST GPR.

7.6 Load Instructions

This group contains instructions used to load words, halfwords or bytes of data from memory. Load instructions must access aligned addresses when a block larger than one byte is accessed.

31 24	23 19	18 14	13 9	8 0
OPC	SIMM	SRC	DST	SIMM
8	5	5	5	9

Field descriptions:

Field	Description
OPC	Operation code of instruction.
DST	Destination GPR.
SRC	Value of source GPR is used as base address.
SIMM	Signed immediate.

7.6.1 LD

Instruction	**LD**
Op. code	0x0E
Syntax	DST = ld [SRC + SIMM]
Semantics	DST ← mem[SRC + (int14)SIMM]
Latency	2

31 24	23 19	18 14	13 9	8 0
0x0E	SIMM	SRC	DST	SIMM
8	5	5	5	9

Example	r1 = ld [r2 + 0]
Description	This instruction loads a 32-bit word from memory. The access address must be aligned to 4 bytes, i.e. the lowest 2 bits of must be zeros.

7.6.2 LDHU

Instruction	**LDHU**
Op. code	0x10
Syntax	DST = ldhu [SRC + SIMM]
Semantics	DST ← mem[SRC + (int14)SIMM]
Latency	2

31 24	23 19	18 14	13 9	8 0
0x10	SIMM	SRC	DST	SIMM
8	5	5	5	9

Example	r1 = ldhu [r2 + 0]
Description	This instruction loads an unsigned half-word from memory. The access address must be aligned to 2 bytes, i.e. the lowest bit of address must be zero.

7.6.3 LDHS

Instruction	**LDHS**
Op. code	0x0F
Syntax	DST = ldhs [SRC + SIMM]
Semantics	DST ← (int32)(int16)mem[SRC + (int14)SIMM]
Latency	2

31 24	23 19	18 14	13 9	8 0
0x0F	SIMM	SRC	DST	SIMM
8	5	5	5	9

Example	r1 = ldhs [r2 + 0]
Description	This instruction loads a signed half-word from memory. The access address must be aligned to 2 bytes, i.e. the lowest bit of address must be zero.

7.6.4 LDBU

Instruction	**LDBU**
Op. code	0x12
Syntax	DST = ldbu [SRC + SIMM]
Semantics	DST ← mem[SRC + (int14)SIMM]
Latency	2

31 24	23 19	18 14	13 9	8 0
0x18	SIMM	SRC	DST	SIMM
8	5	5	5	9

Example	r1 = ldbu [r2 + 0]
Description	This instruction loads an unsigned byte from memory. The access address does not have to be aligned.

7.6.5 LDBS

Instruction	**LDBS**
Op. code	0x13
Syntax	DST = ldbs [SRC + SIMM]
Semantics	DST ← (int32)(int8)mem[SRC + (int14)SIMM]
Latency	2

31 24	23 19	18 14	13 9	8 0
0x13	SIMM	SRC	DST	SIMM
8	5	5	5	9

Example	r1 = ldbs [r2 + 0]
Description	This instruction loads a signed byte from memory. The access address does not have to be aligned.

7.7 Store Instructions

This group contains instructions used to store words, half-words or bytes of data to memory.

31 24	23 19	18 14	13 9	8 0
OPC	SIMM	SRC_1	SRC_2	SIMM
8	5	5	5	9

Field descriptions:

Field	Description
OPC	Operation code of instruction.
SRC_1	Source GPR used as base address.
SRC_2	Source GPR from which the value is stored to the memory.
SIMM	Signed immediate.

7.7.1 ST

Instruction	**ST**
Op. code	0x13
Syntax	st SRC_2, [SRC_1 + SIMM]
Semantics	mem[SRC_1+ (int14)SIMM] ← SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x13	SIMM	SRC_1	SRC_2	SIMM
8	5	5	5	9

Example	st r1, [r2 + 0]
Description	This instruction stores a word to memory. The access address must be aligned to 4 bytes, i.e. the lowest 2 bits of access address must be zero.

7.7.2 STB

Instruction	**STB**
Op. code	0x15

Syntax	stb SRC_2, [SRC_1 + SIMM]
Semantics	mem[SRC_1+ (int14)SIMM] ← (uint8)SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x13	SIMM	SRC_1	SRC_2	SIMM
8	5	5	5	9

Example	stb r1, [r2 + 0]
Description	This instruction stores a byte to memory. The access address does not have to be aligned.

7.7.3 STH

Instruction	**STH**
Op. code	0x14
Syntax	sth SRC_2, [SRC_1 + SIMM]
Semantics	mem[SRC_1+ (int14)SIMM] ← (uint16)SRC_2
Latency	1

31 24	23 19	18 14	13 9	8 0
0x14	SIMM	SRC_1	SRC_2	SIMM
8	5	5	5	9

Example	sth r1, [r2 + 0]
Description	This instruction stores a half-word to memory. The access address must be aligned to 2 bytes, i.e. the lowest bit of address must be zero.

7.8 Jump and Call instructions

Instructions in this group are used to modify the control flow of the program. These instructions use immediate operands as absolute addresses to modify the content of the program counter.

31 24	23 0
OPC	ABS_ADDR
8	24

Field descriptions:

Field	Description
OPC	Operation code of instruction.
ABS_ADDR	Absolute address - unsigned immediate.

7.8.1 JUMP

Instruction	**JUMP**
Op. code	0x16
Syntax	jump
Semantics	PC ← (uint24)ABS_ADDR
Latency	1

31	**24**	**23**	**0**
0x16		ABS_ADDR	
8		24	

Example	jump $label
Description	New program counter value is set to an absolute address.

7.8.2 CALL

Instruction	**CALL**
Op. code	0x17
Syntax	call ABS_ADDR
Semantics	R3 ← PC + 4; PC ← (uint24)ABS_ADDR
Latency	1

31	**24**	**23**	**0**
0x17		ABS_ADDR	
8		24	

Example	call $main
Description	Return address is stored to GPR R3. Program counter is set to absolute address and the next instruction to be executed will be fetched from the updated address in the program counter.

7.9 Indirect Jump and Call Instructions

This group of instructions also modifies the program flow by modifying the program counter by values stored in register.

31	**24**	**23**	**14**	**13**	**9**	**8**	**0**
OPC		UNUSED		SRC		UNUSED	
8		10		5		9	

7.9.1 JUMP

Instruction	**JUMP**
Op. code	0x18
Syntax	jump SRC
Semantics	PC ← SRC
Latency	1

31 24	23 14	13 9	8 0
0x18	0x0	SRC	0x0
8	10	5	9

Example	jump r1
Description	Program counter is set to address stored in register SRC GPR.

7.9.2 CALL

Instruction	**CALL**
Op. code	0x19
Syntax	call SRC
Semantics	R3 ← PC + 4; PC ← SRC
Latency	1

31 24	23 14	13 9	8 0
0x19	0x0	SRC	0x0
8	10	5	9

Example	call r1
Description	Return address is stored to GPR R3. Program counter is set to address stored in SRC_2 GPR and the next instruction to be executed will be fetched from the updated address in the program counter.

7.10 Conditional Jump Instructions

Instructions in this group provide conditional relative jumps usually used to jump to instructions in the same function.

31 24	23 19	18 14	13 9	8 0
OPC	REL_ADDR	SRC	UNUSED	REL_ADDR
8	5	5	5	9

Field descriptions:

Field	Description
OPC	Operation code of instruction.
SRC	Condition GPR.
UNUSED	Unused bits, filled with zeros.
REL_ADDR	Relative address of jump destination. A signed value of the address of label is read by the assembler. Value of program counter is subtracted and the result is stored in binary coding.

7.10.1 JUMPZ

Instruction	**JUMPZ**
Op. code	0x22
Syntax	jumpz SRC, REL_ADDR
Semantics	if (SRC == 0) PC ← PC + (int14)REL_ADDR
Latency	1

31 24	23 19	18 14	13 9	8 0
0x22	REL_ADDR	SRC	UNUSED	REL_ADDR
8	5	5	5	9

Example	jumpz r1, $label
Description	If the value of GPR SRC equals zero, the value of a relative address is added to the current program counter value and jump is performed. If the condition is not met the program counter is incremented to address of next instruction following the conditional JUMPZ instruction. When instruction is fetched the program counter is immediately incremented to address of the next instruction.

7.10.2 JUMPNZ

Instruction	**JUMPNZ**
Op. code	0x23
Syntax	jumpnz SRC, REL_ADDR16
Semantics	if (SRC != 0) PC ← PC + (int14)REL_ADDR
Latency	1

31 24	23 19	18 14	13 9	8 0
0x23	REL_ADDR	SRC	UNUSED	REL_ADDR
8	5	5	5	9

Example	jumpnz r1, $label
Description	If the value of GPR SRC is other than zero, the value of a relative address is added to the current program counter value and jump is performed. If the condition is not met the program counter is incremented to the address of the next instruction following the conditional JUMPNZ instruction. When instruction is fetched the program counter is immediately incremented to address of the next instruction.

8 INSTRUCTION SET LISTINGS

OPCODE	INSTRUCTION PARAMETERS				SYNTAX
0x00	UNUSED:24				nop
0x01	UNUSED:24				halt
0x02	SIMM:5	DST:5	SIMM:14		DST = movsi SIMM
0x03	SIMM:5	DST:5	SIMM:14		DST = movhi SIMM
0x04	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = add SRC_1, SRC_2
0x05	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = sub SRC_1, SRC_2
0x06	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = mul SRC_1, SRC_2
0x07	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = and SRC_1, SRC_2
0x08	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = or SRC_1, SRC_2
0x09	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = xor SRC_1, SRC_2
0x0A	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = sll SRC_1, SRC_2
0x0B	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = srl SRC_1, SRC_2
0x0C	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = sra SRC_1, SRC_2
0x0D	SIMM:5	SRC:5	DST:5	SIMM:9	DST = ld [SRC+SIMM]
0x0E	SIMM:5	SRC:5	DST:5	SIMM:9	DST = ldhs [SRC+SIMM]
0x0F	SIMM:5	SRC:5	DST:5	SIMM:9	DST = ldhu [SRC+SIMM]
0x10	SIMM:5	SRC:5	DST:5	SIMM:9	DST = ldbs [SRC+SIMM]
0x11	SIMM:5	SRC:5	DST:5	SIMM:9	DST = ldbu [SRC+SIMM]
0x12	SIMM:5	SRC_1:5	SRC_2:5	SIMM:9	st SRC_2, [SRC_1+SIMM]
0x13	SIMM:5	SRC_1:5	SRC_2:5	SIMM:9	sth SRC_2, [SRC_1+SIMM]
0x14	SIMM:5	SRC_1:5	SRC_2:5	SIMM:9	stb SRC_2, [SRC_1+SIMM]
0x15	ABS_ADDR:24				jump ABS_ADDR
0x16	ABS_ADDR:24				call ABS_ADDR
0x17	UNUSED:10		SRC:5	UNUSED:9	jump SRC
0x18	UNUSED:10		SRC:5	UNUSED:9	call SRC
0x19	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = eq SRC_1, SRC_2
0x1A	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = neq SRC_1, SRC_2
0x1B	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = slt SRC_1, SRC_2
0x1C	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = ult SRC_1, SRC_2
0x1D	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = sle SRC_1, SRC_2
0x1E	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = ule SRC_1, SRC_2
0x1F	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = movz SRC_1, SRC_2
0x20	DST:5	SRC_1:5	SRC_2:5	UNUSED:9	DST = movnz SRC_1, SRC_2

OPCODE	INSTRUCTION PARAMETERS				SYNTAX
0x21	REL_ADDR:5	SRC:5	UNUSED:5	REL_ADDR:9	jumpz SRC, REL_ADDR
0x22	REL_ADDR:5	SRC:5	UNUSED:5	REL_ADDR:9	jumpnz SRC, REL_ADDR
0x23	SIMM:5	SRC:5	DST:5	SIMM:9	DST = addi SRC, SIMM

9 REVISION HISTORY

Tab. 3: Revision history.

Document Version	Date	Description
1.0	13 Dec 2019	New versioning scheme in accordance with the new documentation style.

10 ANNEX: TABLES, EXAMPLES AND FIGURES

10.1 List of Tables

10.2 List of Examples

10.3 List of Figures