1. Zc* v1.0

1.1. Change history since v0.70.1 (tagged release)

Table 1. Change history

Version	change
v1.0	Ratified Version
v1.0.3-1	Replace statement about non-idempotent memory handler completing the sequence (non-normative)
v1.0.3	Add definition of Zce
v1.0.2	Fix Architecture Review Committee feedback on instruction formats
v1.0.1	Post public review fixes: Add instruction formats (issue 192). Clarify that Zcmt/Zcmp are for embedded CPUs (issue 190). Fix some typos.
v1.0.0-RC5.7	Add Zcb description and fix some typos. PUBLIC REVIEW REVISION.
v1.0.0-RC5.6	Remove Zcmpe which is *not* frozen and is causing confusion
v1.0.0-RC5.5	Following ARC review Adjust the split so we have 224 cm.jalt and 32 cm.jt
v1.0.0-RC5.4	Change wording for dependencies to match arch manual "Zxxx requires Zyyy" changed to "Zxxx depends on Zyyy"
v1.0.0-RC5.3	Add dependency on Zicsr for Zcmt
v1.0.0-RC5.2	Adjust the split so we have 240 cm.jalt and 16 cm.jt
v1.0.0-RC5.1	Make cm.jt/cm.jalt only valid if JVT.mode=0, and allow different behaviour in the future if JVT.mode>0
v1.0.0-RC5	Revert to cm.jt and cm.jalt encodings, to avoid toolchain and trace problems
v1.0.0-RC4.1	Resolve typographical issues with the document only, no actual changes
v1.0.0-RC4	Release candidate
	Remove Zcmb as benefit is low. Remove cm.jalt, read LSB of jump table entry to determine whether to link

Version	change
v0.70.5	Resolve https://github.com/riscv/riscv-code-size-reduction/issues/163 - jvt.base is WARL and fewer bits than the max can be implemented
v0.70.4	Clarified https://github.com/riscv/riscv-code-size-reduction/issues/159 - Need Zbb and Zba for RV64 and M/ZMmul to get *all* of Zcb
	Resolved https://github.com/riscv/riscv-code-size-reduction/issues/161
	Resolved https://github.com/riscv/riscv-code-size-reduction/issues/160 - Allocated Smstateen bit 2 and added the relevant text
v0.70.3	Added rule that Zcf and Zcmt imply Zca (this text was missing, this is not a spec change: https://github.com/riscv/riscv-code-size-reduction/pull/151)
	Added that Zcf is illegal for RV64, as it contains no instructions (clarification: https://github.com/riscv/riscv-code-size-reduction/issues/149)
	Added push/pop examples in the push/pop section
v0.70.2	Stylistic changes only, removing redundant text.
	Corrected field names on JVT CSR diagram, and fixed synopsis for cm.mvsa01

1.2. Zc* Overview

This extension is Ratified. No changes are allowed. Any desired or needed changes can be the subject of a follow-on new extension. Ratified extensions are never revised. See https://riscv.org/spec-state.

Zc* is a group of extensions which define subsets of the existing C extension (Zca, Zcd, Zcf) and new extensions which only contain 16-bit encodings.

Zcm* all reuse the encodings for *c.fld, c.fsd, c.fldsp, c.fsdsp*.

Table 2. Zc extension overview*

Instruction	Zca	Zcf	Zcd	Zcb	Zcmp	Zcmt
The Zca extension is added as way to refer to instructions in the C extension that do not include the floating-point loads and stores						
C excl. c.f*	yes					

Instruction	Zca	Zcf	Zcd	Zcb	Zcmp	Zcmt
The Zcf extension is added as a way to refer to compressed single-precision floating-point load/stores						
c.flw		yes				
c.flwsp		yes				
c.fsw		yes				
c.fswsp		yes				
The Zcd extension is added as a way to refer to compressed double-precision floating-point load/stores						
c.fld			yes			
c.fldsp			yes			
c.fsd			yes			
c.fsdsp			yes			
Simple operations for use on all architectures						
c.lbu				yes		
c.lh				yes		
c.lhu				yes		
c.sb				yes		
c.sh				yes		
c.zext.b				yes		
c.sext.b				yes		
c.zext.h				yes		
c.sext.h				yes		
c.zext.w				yes		
c.mul				yes		
c.not				yes		
PUSH/POP and double move which overlap with *c.fsdsp*. Complex operations intended for embedded CPUs						
cm.push					yes	
cm.pop					yes	
cm.popret					yes	
cm.popretz					yes	
cm.mva01s					yes	
cm.mvsa01					yes	
Table jump which overlaps with *c.fsdsp*. Complex operations intended for embedded CPUs						

Instruction	Zca	Zcf	Zcd	Zcb	Zcmp	Zcmt
cm.jt						yes
cm.jalt						yes

1.3. Zce

The Zce extension is intended to be used for microcontrollers, and includes all relevant Zc extensions.

- Specifying Zce without F includes Zca, Zcb, Zcmp, Zcmt
- Specifying Zce with F includes Zca, Zcb, Zcmp, Zcmt *and* Zcf

Therefore common ISA strings can be updated as follows to include the relevant Zc extensions, for example:

- RV32IMC becomes RV32IM_Zce
- RV32IMCF becomes RV32IMF_Zce

1.4. Zca

The Zca extension is added as way to refer to instructions in the C extension that do not include the floating-point loads and stores.

Therefore it *excluded* all 16-bit floating point loads and stores: *c.flw*, *c.flwsp*, *c.fsw*, *c.fswsp*, *c.fld*, *c.fldsp*, *c.fsd*, *c.fsdsp*.

NOTE	the C extension only includes F/D instructions when D and F are also specified

1.5. Zcf (RV32 only)

Zcf is the existing set of compressed single precision floating point loads and stores: *c.flw*, *c.flwsp*, *c.fsw*, *c.fswsp*.

Zcf is only relevant to RV32, it cannot be specified for RV64.

The Zcf extension depends on the Zca extension.

1.6. Zcd

Zcd is the existing set of compressed double precision floating point loads and stores: *c.fld*, *c.fldsp*, *c.fsd*, *c.fsdsp*.

The Zcd extension depends on the Zca extension.

1.7. Zcb

Zcb has simple code-size saving instructions which are easy to implement on all CPUs.

All proposed encodings are currently reserved for all architectures, and have no conflicts with any existing extensions.

NOTE	Zcb can be implemented on *any* CPU as the instructions are 16-bit versions of existing 32-bit instructions from the application class profile.

The Zcb extension depends on the Zca extension.

As shown on the individual instruction pages, many of the instructions in Zcb depend upon another extension being implemented. For example, *c.mul* is only implemented if M or Zmmul is implemented, and *c.sext.b* is only implemented if Zbb is implemented.

The *c.mul* encoding uses the CA register format along with other instructions such as *c.sub*, *c.xor* etc.

NOTE	*c.sext.w* is a pseudo-instruction for *c.addiw rd, 0* (RV64)

RV32	RV64	Mnemonic	Instruction
yes	yes	c.lbu *rd'*, uimm(*rs1'*)	Load unsigned byte, 16-bit encoding
yes	yes	c.lhu *rd'*, uimm(*rs1'*)	Load unsigned halfword, 16-bit encoding
yes	yes	c.lh *rd'*, uimm(*rs1'*)	Load signed halfword, 16-bit encoding
yes	yes	c.sb *rs2'*, uimm(*rs1'*)	Store byte, 16-bit encoding
yes	yes	c.sh *rs2'*, uimm(*rs1'*)	Store halfword, 16-bit encoding
yes	yes	c.zext.b *rsd'*	Zero extend byte, 16-bit encoding
yes	yes	c.sext.b *rsd'*	Sign extend byte, 16-bit encoding
yes	yes	c.zext.h *rsd'*	Zero extend halfword, 16-bit encoding
yes	yes	c.sext.h *rsd'*	Sign extend halfword, 16-bit encoding
	yes	c.zext.w *rsd'*	Zero extend word, 16-bit encoding
yes	yes	c.not *rsd'*	Bitwise not, 16-bit encoding
yes	yes	c.mul *rsd'*, *rs2'*	Multiply, 16-bit encoding

1.8. Zcmp

The Zcmp extension is a set of instructions which may be executed as a series of existing 32-bit RISC-V instructions.

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with Zcd, which is included when C and D extensions are both present.

NOTE	Zcmp is primarily targeted at embedded class CPUs due to implementation complexity. Additionally, it is not compatible with architecture class profiles.

The Zcmp extension depends on the Zca extension.

The PUSH/POP assembly syntax uses several variables, the meaning of which are:

- *reg_list* is a list containing 1 to 13 registers (ra and 0 to 12 s registers)
  - valid values: {ra}, {ra, s0}, {ra, s0-s1}, {ra, s0-s2}, ..., {ra, s0-s8}, {ra, s0-s9}, {ra, s0-s11}
  - note that {ra, s0-s10} is *not* valid, giving 12 lists not 13 for better encoding
- *stack_adj* is the total size of the stack frame.
  - valid values vary with register list length and the specific encoding, see the instruction pages for details.

RV32	RV64	Mnemonic	Instruction
yes	yes	cm.push *{reg_list}, -stack_adj*	Create stack frame: push registers, allocate additional stack space.
yes	yes	cm.pop *{reg_list}, stack_adj*	Pop registers, deallocate stack frame.
yes	yes	cm.popret *{reg_list}, stack_adj*	Pop registers, deallocate stack frame, return.
yes	yes	cm.popretz *{reg_list}, stack_adj*	Pop registers, deallocate stack frame, return zero.
yes	yes	cm.mva01s *rs1', rs2'*	Move two s0-s7 registers into a0-a1
yes	yes	cm.mvsa01 *r1s', r2s'*	Move a0-a1 into two different s0-s7 registers

1.9. Zcmt

Zcmt adds the table jump instructions and also adds the JVT CSR. The JVT CSR requires a state enable if Smstateen is implemented. See JVT CSR, table jump base vector and control register for details.

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with Zcd, which is included when C and D extensions are both present.

NOTE	Zcmt is primarily targeted at embedded class CPUs due to implementation complexity. Additionally, it is not compatible with architecture class profiles.

The Zcmt extension depends on the Zca and Zicsr extensions.

RV32	RV64	Mnemonic	Instruction
yes	yes	cm.jt *index*	Jump via table
yes	yes	cm.jalt *index*	Jump and link via table

1.10. Zc instruction formats

Several instructions in this specification use the following new instruction formats.

Format	instructions	15:10	9	8	7	6	5	4	3	2	1	0
CLB	c.lbu	funct6	rs1'			uimm		rd'			op	
CSB	c.sb	funct6	rs1'			uimm		rs2'			op	
CLH	c.lhu, c.lh	funct6	rs1'			funct1	uimm	rd'			op	
CSH	c.sh	funct6	rs1'			funct1	uimm	rs2'			op	
CU	c.[sz]ext.*, c.not	funct6	rd'/rs1'			funct5					op	
CMMV	cm.mvsa01 cm.mva01s	funct6	r1s'			funct2		r2s'			op	
CMJT	cm.jt cm.jalt	funct6	index								op	
CMPP	cm.push*, cm.pop*	funct6	funct2		urlist				spimm		op	

NOTE	c.mul uses the existing CA format

2. Zcb instructions

2.1. c.lbu

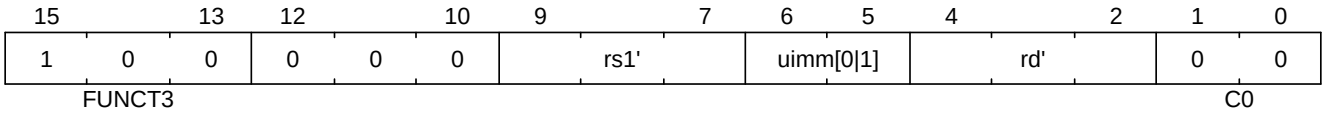
Synopsis

Load unsigned byte, 16-bit encoding

Mnemonic

c.lbu *rd'*, *uimm*(*rs1'*)

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = encoding[6];
```

Description

This instruction loads a byte from the memory address formed by adding *rs1'* to the zero extended immediate *uimm*. The resulting byte is zero extended to XLEN bits and is written to *rd'*.

NOTE *rd'* and *rs1'* are from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

[insns-lbu]

Operation

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

X(rdc) = EXTZ(mem[X(rs1c)+EXTZ(uimm)][7..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.2. c.lhu

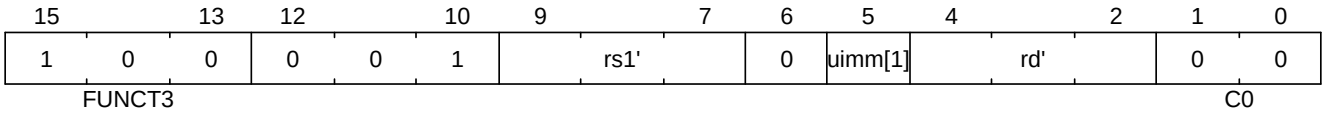
Synopsis

Load unsigned halfword, 16-bit encoding

Mnemonic

c.lhu *rd'*, *uimm*(*rs1'*)

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = 0;
```

Description

This instruction loads a halfword from the memory address formed by adding *rs1'* to the zero extended immediate *uimm*. The resulting halfword is zero extended to XLEN bits and is written to *rd'*.

NOTE	*rd'* and *rs1'* are from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

[insns-lhu]

Operation

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

X(rdc) = EXTZ(load_mem[X(rs1c)+EXTZ(uimm)][15..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.3. c.lh

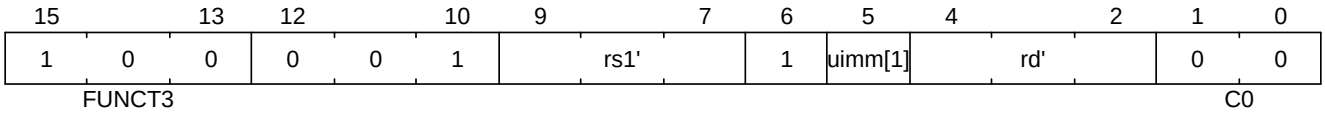
Synopsis

Load signed halfword, 16-bit encoding

Mnemonic

c.lh *rd'*, *uimm*(*rs1'*)

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = 0;
```

Description

This instruction loads a halfword from the memory address formed by adding *rs1'* to the zero extended immediate *uimm*. The resulting halfword is sign extended to XLEN bits and is written to *rd'*.

NOTE	*rd'* and *rs1'* are from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

[insns-lh]

Operation

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

X(rdc) = EXTS(load_mem[X(rs1c)+EXTZ(uimm)][15..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.4. c.sb

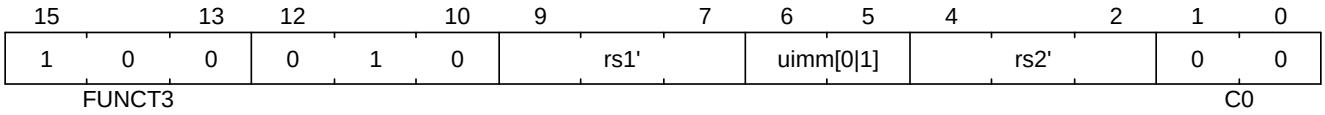
Synopsis

Store byte, 16-bit encoding

Mnemonic

c.sb *rs2'*, *uimm*(*rs1'*)

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = encoding[6];
```

Description

This instruction stores the least significant byte of *rs2'* to the memory address formed by adding *rs1'* to the zero extended immediate *uimm*.

NOTE *rs1'* and *rs2'* are from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

[insns-sb]

Operation

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

mem[X(rs1c)+EXTZ(uimm)][7..0] = X(rs2c)
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.5. c.sh

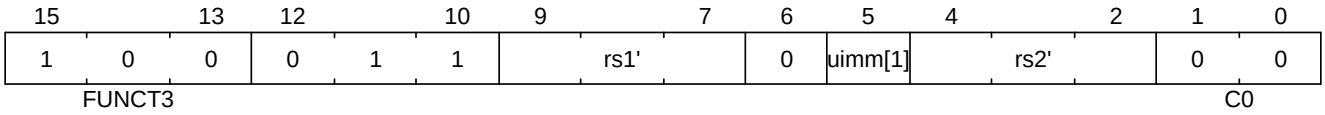
Synopsis

Store halfword, 16-bit encoding

Mnemonic

c.sh *rs2'*, *uimm*(*rs1'*)

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:2] = 0;
uimm[1]    = encoding[5];
uimm[0]    = 0;
```

Description

This instruction stores the least significant halfword of *rs2'* to the memory address formed by adding *rs1'* to the zero extended immediate *uimm*.

NOTE *rs1'* and *rs2'* are from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

[insns-sh]

Operation

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

mem[X(rs1c)+EXTZ(uimm)][15..0] = X(rs2c)
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.6. c.zext.b

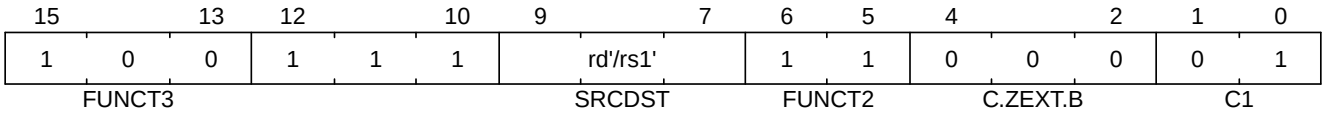
Synopsis

Zero extend byte, 16-bit encoding

Mnemonic

c.zext.b *rd'/rs1'*

Encoding (RV32, RV64)



Description

This instruction takes a single source/destination operand. It zero-extends the least-significant byte of the operand to XLEN bits by inserting zeros into all of the bits more significant than 7.

NOTE *rd'/rs1'* is from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

```
andi rd'/rs1', rd'/rs1', 0xff
```

NOTE The SAIL module variable for *rd'/rs1'* is called *rsdc*.

Operation

```
X(rsdc) = EXTZ(X(rsdc)[7..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.7. c.sext.b

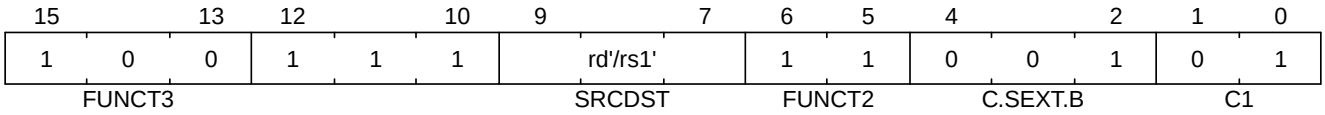
Synopsis

Sign extend byte, 16-bit encoding

Mnemonic

c.sext.b *rd'/rs1'*

Encoding (RV32, RV64)



Description

This instruction takes a single source/destination operand. It sign-extends the least-significant byte in the operand to XLEN bits by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.

NOTE *rd'/rs1'* is from the standard 8-register set x8-x15.

Prerequisites

Zbb is also required.

32-bit equivalent

[insns-sext_b] from Zbb

NOTE The SAIL module variable for *rd'/rs1'* is called *rsdc*.

Operation

```
X(rsdc) = EXTS(X(rsdc)[7..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.8. c.zext.h

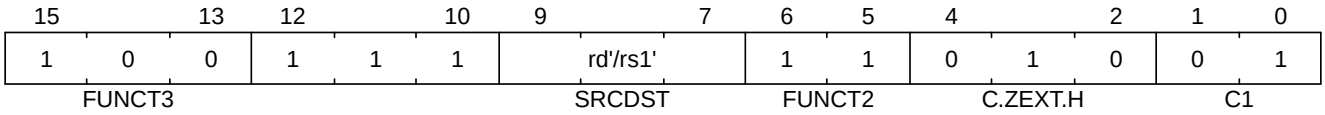
Synopsis

Zero extend halfword, 16-bit encoding

Mnemonic

c.zext.h *rd'/rs1'*

Encoding (RV32, RV64)



Description

This instruction takes a single source/destination operand. It zero-extends the least-significant halfword of the operand to XLEN bits by inserting zeros into all of the bits more significant than 15.

NOTE *rd'/rs1'* is from the standard 8-register set x8-x15.

Prerequisites

Zbb is also required.

32-bit equivalent

[insns-zext_h] from Zbb

NOTE The SAIL module variable for *rd'/rs1'* is called *rsdc*.

Operation

```
X(rsdc) = EXTZ(X(rsdc)[15..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.9. c.sext.h

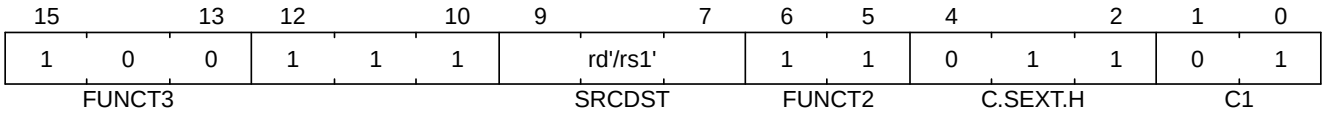
Synopsis

Sign extend halfword, 16-bit encoding

Mnemonic

c.sext.h *rd'/rs1'*

Encoding (RV32, RV64)



Description

This instruction takes a single source/destination operand. It sign-extends the least-significant halfword in the operand to XLEN bits by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits.

NOTE *rd'/rs1'* is from the standard 8-register set x8-x15.

Prerequisites

Zbb is also required.

32-bit equivalent

[insns-sext_h] from Zbb

NOTE The SAIL module variable for *rd'/rs1'* is called *rsdc*.

Operation

```
X(rsdc) = EXTS(X(rsdc)[15..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.10. c.zext.w

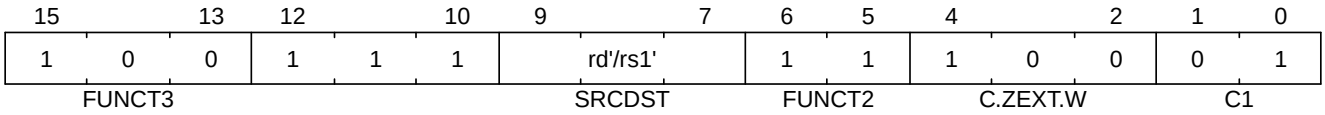
Synopsis

Zero extend word, 16-bit encoding

Mnemonic

c.zext.w *rd'/rs1'*

Encoding (RV64)



Description

This instruction takes a single source/destination operand. It zero-extends the least-significant word of the operand to XLEN bits by inserting zeros into all of the bits more significant than 31.

NOTE *rd'/rs1'* is from the standard 8-register set x8-x15.

Prerequisites

Zba is also required.

32-bit equivalent

```
add.uw rd'/rs1', rd'/rs1', zero
```

NOTE The SAIL module variable for *rd'/rs1'* is called *rsdc*.

Operation

```
X(rsdc) = EXTZ(X(rsdc)[31..0]);
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.11. c.not

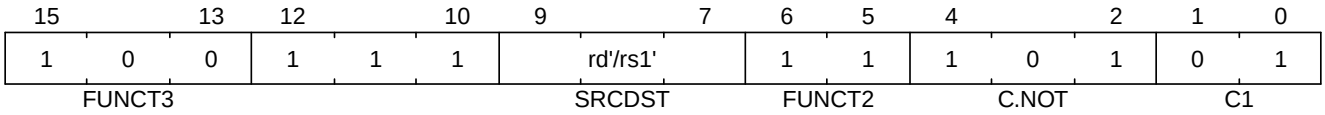
Synopsis

Bitwise not, 16-bit encoding

Mnemonic

c.not *rd'/rs1'*

Encoding (RV32, RV64)



Description

This instruction takes the one's complement of *rd'/rs1'* and writes the result to the same register.

NOTE *rd'/rs1'* is from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

```
xori rd'/rs1', rd'/rs1', -1
```

NOTE The SAIL module variable for *rd'/rs1'* is called *rsdc*.

Operation

```
X(rsdc) = X(rsdc) XOR -1;
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

2.12. c.mul

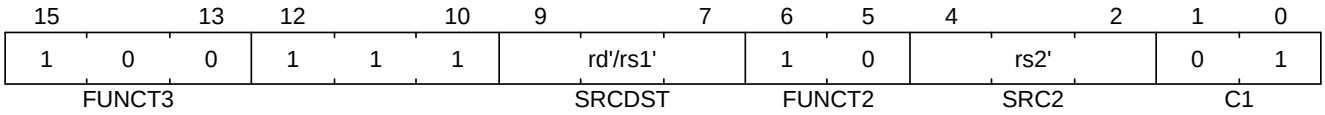
Synopsis

Multiply, 16-bit encoding

Mnemonic

c.mul *rsd'*, *rs2'*

Encoding (RV32, RV64)



Description

This instruction multiplies XLEN bits of the source operands from *rsd'* and *rs2'* and writes the lowest XLEN bits of the result to *rsd'*.

NOTE *rd'/rs1'* and *rs2'* are from the standard 8-register set x8-x15.

Prerequisites

M or Zmmul must be configured.

32-bit equivalent

[insns-mul]

NOTE The SAIL module variable for *rd'/rs1'* is called *rsdc*, and for *rs2'* is called *rs2c*.

Operation

```
let result_wide = to_bits(2 * sizeof(xlen), signed(X(rsdc)) * signed(X(rs2c)));
X(rsdc) = result_wide[(sizeof(xlen) - 1) .. 0];
```

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v1.0	Ratified

3. PUSH/POP register instructions

These instructions are collectively referred to as PUSH/POP:

- Create stack frame: push registers, allocate additional stack space.
- Pop registers, deallocate stack frame.
- Pop registers, deallocate stack frame, return.
- Pop registers, deallocate stack frame, return zero.

The term PUSH refers to *cm.push*.

The term POP refers to *cm.pop*.

The term POPRET refers to *cm.popret and cm.popretz*.

Common details for these instructions are in this section.

3.1. PUSH/POP functional overview

PUSH, POP, POPRET are used to reduce the size of function prologues and epilogues.

1. The PUSH instruction
    - adjusts the stack pointer to create the stack frame
    - pushes (stores) the registers specified in the register list to the stack frame
2. The POP instruction
    - pops (loads) the registers in the register list from the stack frame
    - adjusts the stack pointer to destroy the stack frame
3. The POPRET instructions
    - pop (load) the registers in the register list from the stack frame
    - *cm.popretz* also moves zero into *a0* as the return value
    - adjust the stack pointer to destroy the stack frame
    - execute a *ret* instruction to return from the function

3.2. Example usage

This example gives an illustration of the use of PUSH and POPRET.

The function *processMarkers* in the EMBench benchmark picojpeg in the following file on github: libpicojpeg.c

The prologue and epilogue compile with GCC10 to:

```
   0001098a <processMarkers>:
   1098a:       711d                    addi    sp,sp,-96 ;#cm.push(1)
   1098c:       c8ca                    sw      s2,80(sp) ;#cm.push(2)
   1098e:       c6ce                    sw      s3,76(sp) ;#cm.push(3)
   10990:       c4d2                    sw      s4,72(sp) ;#cm.push(4)
   10992:       ce86                    sw      ra,92(sp) ;#cm.push(5)
   10994:       cca2                    sw      s0,88(sp) ;#cm.push(6)
   10996:       caa6                    sw      s1,84(sp) ;#cm.push(7)
   10998:       c2d6                    sw      s5,68(sp) ;#cm.push(8)
   1099a:       c0da                    sw      s6,64(sp) ;#cm.push(9)
   1099c:       de5e                    sw      s7,60(sp) ;#cm.push(10)
   1099e:       dc62                    sw      s8,56(sp) ;#cm.push(11)
   109a0:       da66                    sw      s9,52(sp) ;#cm.push(12)
   109a2:       d86a                    sw      s10,48(sp);#cm.push(13)
   109a4:       d66e                    sw      s11,44(sp);#cm.push(14)
...
   109f4:       4501                    li      a0,0      ;#cm.popretz(1)
   109f6:       40f6                    lw      ra,92(sp) ;#cm.popretz(2)
   109f8:       4466                    lw      s0,88(sp) ;#cm.popretz(3)
   109fa:       44d6                    lw      s1,84(sp) ;#cm.popretz(4)
   109fc:       4946                    lw      s2,80(sp) ;#cm.popretz(5)
   109fe:       49b6                    lw      s3,76(sp) ;#cm.popretz(6)
   10a00:       4a26                    lw      s4,72(sp) ;#cm.popretz(7)
   10a02:       4a96                    lw      s5,68(sp) ;#cm.popretz(8)
   10a04:       4b06                    lw      s6,64(sp) ;#cm.popretz(9)
   10a06:       5bf2                    lw      s7,60(sp) ;#cm.popretz(10)
   10a08:       5c62                    lw      s8,56(sp) ;#cm.popretz(11)
   10a0a:       5cd2                    lw      s9,52(sp) ;#cm.popretz(12)
   10a0c:       5d42                    lw      s10,48(sp);#cm.popretz(13)
   10a0e:       5db2                    lw      s11,44(sp);#cm.popretz(14)
   10a10:       6125                    addi    sp,sp,96  ;#cm.popretz(15)
   10a12:       8082                    ret               ;#cm.popretz(16)
```

with the GCC option *-msave-restore* the output is the following:

```
0001080e <processMarkers>:
   1080e:       73a012ef                jal     t0,11f48 <__riscv_save_12>
   10812:       1101                    addi    sp,sp,-32
...
   10862:       4501                    li      a0,0
   10864:       6105                    addi    sp,sp,32
   10866:       71e0106f                j       11f84 <__riscv_restore_12>
```

with PUSH/POPRET this reduces to

```
0001080e <processMarkers>:
   1080e:       b8fa                    cm.push    {ra,s0-s11},-96
...
   10866:       bcfa                    cm.popretz {ra,s0-s11}, 96
```

The prologue / epilogue reduce from 60-bytes in the original code, to 14-bytes with *-msave-restore*, and to 4-bytes with PUSH and POPRET. As well as reducing the code-size PUSH and POPRET eliminate the branches from calling the millicode *save/restore* routines and so may also perform better.

NOTE The calls to *<riscv_save_0>/<riscv_restore_0>* become 64-bit when the target functions are out of the ±1MB range, increasing the prologue/epilogue size to 22-bytes.

NOTE POP is typically used in tail-calling sequences where *ret* is not used to return to *ra* after destroying the stack frame.

3.2.1. Stack pointer adjustment handling

The instructions all automatically adjust the stack pointer by enough to cover the memory required for the registers being saved or restored. Additionally the *spimm* field in the encoding allows the stack pointer to be adjusted in additional increments of 16-bytes. There is only a small restricted range available in the encoding; if the range is insufficient then a separate *c.addi16sp* can be used to increase the range.

3.2.2. Register list handling

There is no support for the *{ra, s0-s10}* register list without also adding *s11*. Therefore the *{ra, s0-s11}* register list must be used in this case.

3.3. PUSH/POP Fault handling

Correct execution requires that *sp* refers to idempotent memory (also see Non-idempotent memory handling), because the core must be able to handle traps detected during the sequence. The entire

PUSH/POP sequence is re-executed after returning from the trap handler, and multiple traps are possible during the sequence.

If a trap occurs during the sequence then *xEPC* is updated with the PC of the instruction, *xTVAL* (if not read-only-zero) updated with the bad address if it was an access fault and *xCAUSE* updated with the type of trap.

NOTE	It is implementation defined whether interrupts can also be taken during the sequence execution.

3.4. Software view of execution

3.4.1. Software view of the PUSH sequence

From a software perspective the PUSH sequence appears as:

- A sequence of stores writing the bytes required by the pseudo-code
  - The bytes may be written in any order.
  - The bytes may be grouped into larger accesses.
  - Any of the bytes may be written multiple times.
- A stack pointer adjustment

NOTE	If an implementation allows interrupts during the sequence, and the interrupt handler uses *sp* to allocate stack memory, then any stores which were executed before the interrupt may be overwritten by the handler. This is safe because the memory is idempotent and the stores will be re-executed when execution resumes.

The stack pointer adjustment must only be committed only when it is certain that the entire PUSH instruction will commit.

Stores may also return imprecise faults from the bus. It is platform defined whether the core implementation waits for the bus responses before continuing to the final stage of the sequence, or handles errors responses after completing the PUSH instruction.

For example:

```
cm.push  {ra, s0-s5}, -64
```

Appears to software as:

```
# any bytes from sp-1 to sp-28 may be written multiple times before
# the instruction completes therefore these updates may be visible in
# the interrupt/exception handler below the stack pointer
sw  s5, -4(sp)
sw  s4, -8(sp)
sw  s3,-12(sp)
sw  s2,-16(sp)
sw  s1,-20(sp)
sw  s0,-24(sp)
sw  ra,-28(sp)

# this must only execute once, and will only execute after all stores
# completed without any precise faults, therefore this update is only
# visible in the interrupt/exception handler if cm.push has completed
addi sp, sp, -64
```

3.4.2. Software view of the POP/POPRET sequence

From a software perspective the POP/POPRET sequence appears as:

- A sequence of loads reading the bytes required by the pseudo-code.
  - The bytes may be loaded in any order.
  - The bytes may be grouped into larger accesses.
  - Any of the bytes may be loaded multiple times.
- A stack pointer adjustment
- An optional `li a0, 0`
- An optional `ret`

If a trap occurs during the sequence, then any loads which were executed before the trap may update architectural state. The loads will be re-executed once the trap handler completes, so the values will be overwritten. Therefore it is permitted for an implementation to update some of the destination registers before taking a fault.

The optional `li a0, 0`, stack pointer adjustment and optional `ret` must only be committed only when it is certain that the entire POP/POPRET instruction will commit.

For POPRET once the stack pointer adjustment has been committed the `ret` must execute.

For example:

```
cm.popretz {ra, s0-s3}, 32;
```

Appears to software as:

```
# any or all of these load instructions may execute multiple times
# therefore these updates may be visible in the interrupt/exception handler
lw   s3, 28(sp)
lw   s2, 24(sp)
lw   s1, 20(sp)
lw   s0, 16(sp)
lw   ra, 12(sp)

# these must only execute once, will only execute after all loads
# complete successfully all instructions must execute atomically
# therefore these updates are not visible in the interrupt/exception handler
li a0, 0
addi sp, sp, 32
ret
```

3.5. Non-idempotent memory handling

An implementation may have a requirement to issue a PUSH/POP instruction to non-idempotent memory.

If the core implementation does not support PUSH/POP to non-idempotent memories, the core may use an idempotency PMA to detect it and take a load (POP/POPRET) or store (PUSH) access fault exception in order to avoid unpredictable results.

Software should only use these instructions on non-idempotent memory regions when software can tolerate the required memory accesses being issued repeatedly in the case that they cause exceptions.

3.6. Example RV32I PUSH/POP sequences

The examples are included show the load/store series expansion and the stack adjustment. Examples of *cm.popret* and *cm.popretz* are not included, as the difference in the expanded sequence from *cm.pop* is trivial in all cases.

3.6.1. cm.push {ra, s0-s2}, -64

Encoding: *rlist*=7, *spimm*=3

expands to:

```
sw  s2,  -4(sp);
sw  s1,  -8(sp);
sw  s0, -12(sp);
sw  ra, -16(sp);
addi sp, sp, -64;
```

3.6.2. cm.push {ra, s0-s11}, -112

Encoding: *rlist*=15, *spimm*=3

expands to:

```
sw  s11,  -4(sp);
sw  s10,  -8(sp);
sw  s9,  -12(sp);
sw  s8,  -16(sp);
sw  s7,  -20(sp);
sw  s6,  -24(sp);
sw  s5,  -28(sp);
sw  s4,  -32(sp);
sw  s3,  -36(sp);
sw  s2,  -40(sp);
sw  s1,  -44(sp);
sw  s0,  -48(sp);
sw  ra,  -52(sp);
addi sp, sp, -112;
```

3.6.3. cm.pop {ra}, 16

Encoding: *rlist*=4, *spimm*=0

expands to:

```
lw   ra, 12(sp);
addi sp, sp, 16;
```

3.6.4. cm.pop {ra, s0-s3}, 48

Encoding: *rlist*=8, *spimm*=1

expands to:

```
lw   s3, 44(sp);
lw   s2, 40(sp);
lw   s1, 36(sp);
lw   s0, 32(sp);
lw   ra, 28(sp);
addi sp, sp, 48;
```

3.6.5. cm.pop {ra, s0-s4}, 64

Encoding: *rlist*=9, *spimm*=2

expands to:

```
lw   s4, 60(sp);
lw   s3, 56(sp);
lw   s2, 52(sp);
lw   s1, 48(sp);
lw   s0, 44(sp);
lw   ra, 40(sp);
addi sp, sp, 64;
```

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v1.0	Ratified

3.7. cm.push

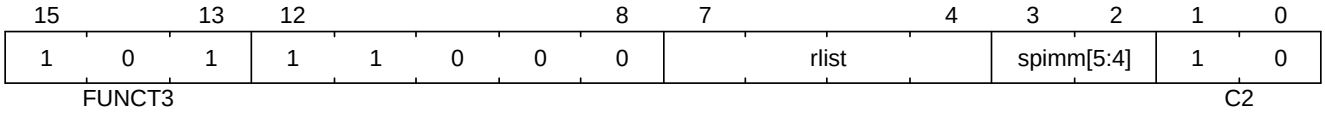
Synopsis

Create stack frame: store ra and 0 to 12 saved registers to the stack frame, optionally allocate additional stack space.

Mnemonic

cm.push *{reg_list}, -stack_adj*

Encoding (RV32, RV64)



NOTE *rlist* values 0 to 3 are reserved for a future EABI variant called *cm.push.e*

Assembly Syntax

```
cm.push {reg_list},  -stack_adj
cm.push {xreg_list}, -stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
```

```
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
```

```
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
 }
```

Description

This instruction pushes (stores) the registers in *reg_list* to the memory below the stack pointer, and then creates the stack frame by decrementing the stack pointer by *stack_adj*, including any additional stack space requested by the value of *spimm*.

NOTE	All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists

Operation

The first section of pseudo-code may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("sw x[i], 0(addr)");
      8:  asm("sd x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudo-code executes atomically, and only executes if the section above completes without any exceptions or interrupts.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

sp-=stack_adj;
```

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v1.0	Ratified

3.8. cm.pop

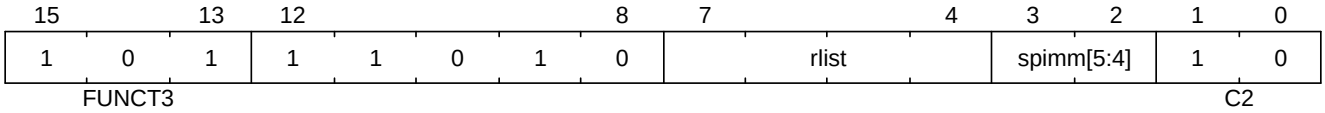
Synopsis

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame.

Mnemonic

cm.pop *{reg_list}, stack_abj*

Encoding (RV32, RV64)



NOTE *rlist* values 0 to 3 are reserved for a future EABI variant called *cm.pop.e*

Assembly Syntax

```
cm.pop {reg_list},  stack_adj
cm.pop {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
```

```
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
```

```
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, and then adjusts the stack pointer by *stack_adj*.

NOTE	All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists

Operation

The first section of pseudo-code may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp+stack_adj-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("lw x[i], 0(addr)");
      8:  asm("ld x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudo-code executes atomically, and only executes if the section above completes without any exceptions or interrupts.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
```

```
sp+=stack_adj;
```

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v1.0	Ratified

3.9. cm.popretz

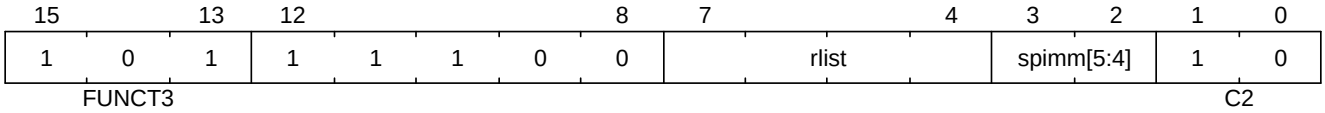
Synopsis

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame, move zero into a0, return to ra.

Mnemonic

cm.popretz *{reg_list}, stack_adj*

Encoding (RV32, RV64)



NOTE *rlist* values 0 to 3 are reserved for a future EABI variant called *cm.popretz.e*

Assembly Syntax

```
cm.popretz {reg_list},  stack_adj
cm.popretz {xreg_list}, stack_adj
```

```
RV32E:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
```

```
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
```

```
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, adjusts the stack pointer by *stack_adj*, moves zero into a0 and then returns to *ra*.

NOTE	All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists

Operation

The first section of pseudo-code may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp+stack_adj-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("lw x[i], 0(addr)");
      8:  asm("ld x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudo-code executes atomically, and only executes if the section above completes without any exceptions or interrupts.

NOTE	The *li a0, 0* **could** be executed more than once, but is included in the atomic section

> for convenience.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

asm("li a0, 0");
sp+=stack_adj;
asm("ret");
```

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v1.0	Ratified

3.10. cm.popret

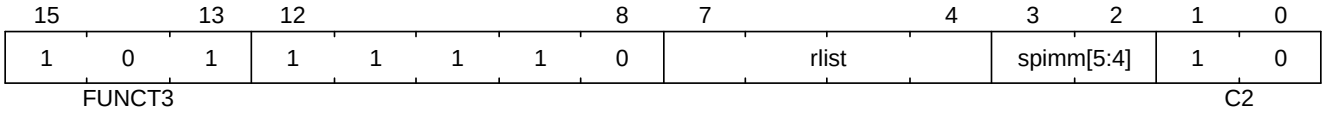
Synopsis

Destroy stack frame: load ra and 0 to 12 saved registers from the stack frame, deallocate the stack frame, return to ra.

Mnemonic

cm.popret *{reg_list}, stack_adj*

Encoding (RV32, RV64)



NOTE *rlist* values 0 to 3 are reserved for a future EABI variant called *cm.popret.e*

Assembly Syntax

```
cm.popret {reg_list},  stack_adj
cm.popret {xreg_list}, stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
```

```
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: reserved();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32E:

stack_adj_base = 16;
Valid values:
stack_adj      = [16|32|48|64];
```

```
RV32I:

switch (rlist) {
  case  4.. 7: stack_adj_base = 16;
  case  8..11: stack_adj_base = 32;
  case 12..14: stack_adj_base = 48;
  case     15: stack_adj_base = 64;
}

Valid values:
switch (rlist) {
  case  4.. 7: stack_adj = [16|32|48| 64];
  case  8..11: stack_adj = [32|48|64| 80];
  case 12..14: stack_adj = [48|64|80| 96];
  case     15: stack_adj = [64|80|96|112];
}
```

```
RV64:

switch (rlist) {
  case  4.. 5: stack_adj_base =  16;
  case  6.. 7: stack_adj_base =  32;
  case  8.. 9: stack_adj_base =  48;
  case 10..11: stack_adj_base =  64;
  case 12..13: stack_adj_base =  80;
  case     14: stack_adj_base =  96;
  case     15: stack_adj_base = 112;
}

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
```

```
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, adjusts the stack pointer by *stack_adj* and then returns to *ra*.

NOTE	All ABI register mappings are for the UABI. An EABI version is planned once the EABI is frozen.

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation

stack_adj_base is the minimum number of bytes, in multiples of 16-byte address increments, required to cover the registers in the list.

spimm is the number of additional 16-byte address increments allocated for the stack frame.

The total stack adjustment represents the total size of the stack frame, which is *stack_adj_base* added to *spimm* scaled by 16, as defined above.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists

Operation

The first section of pseudo-code may be executed multiple times before the instruction successfully completes.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

if (XLEN==32) bytes=4; else bytes=8;

addr=sp+stack_adj-bytes;
for(i in 27,26,25,24,23,22,21,20,19,18,9,8,1)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    switch(bytes) {
      4:  asm("lw x[i], 0(addr)");
      8:  asm("ld x[i], 0(addr)");
    }
    addr-=bytes;
  }
}
```

The final section of pseudo-code executes atomically, and only executes if the section above completes without any exceptions or interrupts.

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
```

```
sp+=stack_adj;
asm("ret");
```

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v1.0	Ratified

3.11. cm.mvsa01

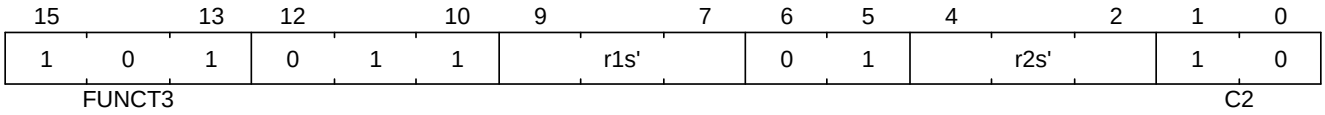
Synopsis

Move a0-a1 into two registers of s0-s7

Mnemonic

cm.mvsa01 *r1s'*, *r2s'*

Encoding (RV32, RV64)



NOTE For the encoding to be legal *r1s'* != *r2s'*.

Assembly Syntax

```
cm.mvsa01 r1s', r2s'
```

Description

This instruction moves *a0* into *r1s'* and *a1* into *r2s'*. *r1s'* and *r2s'* must be different. The execution is atomic, so it is not possible to observe state where only one of *r1s'* or *r2s'* has been updated.

The encoding uses *sreg* number specifiers instead of *xreg* number specifiers to save encoding space. The mapping between them is specified in the pseudo-code below.

NOTE The *s* register mapping is taken from the UABI, and may not match the currently unratified EABI. *cm.mvsa01.e* may be included in the future.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists.

Operation

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
if (RV32E && (r1sc>1 || r2sc>1)) {
  reserved();
}
xreg1 = {r1sc[2:1]>0,r1sc[2:1]==0,r1sc[2:0]};
xreg2 = {r2sc[2:1]>0,r2sc[2:1]==0,r2sc[2:0]};
X[xreg1] = X[10];
X[xreg2] = X[11];
```

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v1.0	Ratified

3.12. cm.mva01s

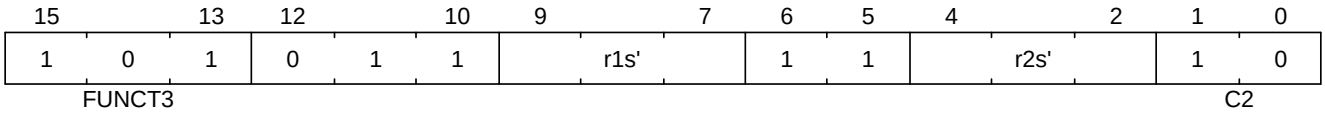
Synopsis

Move two s0-s7 registers into a0-a1

Mnemonic

cm.mva01s *r1s'*, *r2s'*

Encoding (RV32, RV64)



Assembly Syntax

```
cm.mva01s r1s', r2s'
```

Description

This instruction moves *r1s'* into *a0* and *r2s'* into *a1*. The execution is atomic, so it is not possible to observe state where only one of *a0* or *a1* have been updated.

The encoding uses *sreg* number specifiers instead of *xreg* number specifiers to save encoding space. The mapping between them is specified in the pseudo-code below.

NOTE	The *s* register mapping is taken from the UABI, and may not match the currently unratified EABI. *cm.mva01s.e* may be included in the future.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists.

Operation

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.
if (RV32E && (r1sc>1 || r2sc>1)) {
  reserved();
}
xreg1 = {r1sc[2:1]>0,r1sc[2:1]==0,r1sc[2:0]};
xreg2 = {r2sc[2:1]>0,r2sc[2:1]==0,r2sc[2:0]};
X[10] = X[xreg1];
X[11] = X[xreg2];
```

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v1.0	Ratified

4. Table Jump Overview

cm.jt (Jump via table) and *cm.jalt* (Jump and link via table) are referred to as table jump.

Table jump uses a 256-entry XLEN wide table in instruction memory to contain function addresses. The table must be a minimum of 64-byte aligned.

Table entries follow the current data endianness. This is different from normal instruction fetch which is always little-endian.

cm.jt and *cm.jalt* encodings index the table, giving access to functions within the full XLEN wide address space.

This is used as a form of dictionary compression to reduce the code size of *jal* / *auipc+jalr* / *jr* / *auipc+jr* instructions.

Table jump allows the linker to replace the following instruction sequences with a *cm.jt* or *cm.jalt* encoding, and an entry in the table:

- 32-bit *j* calls
- 32-bit *jal* ra calls
- 64-bit *auipc+jr* calls to fixed locations
- 64-bit *auipc+jalr ra* calls to fixed locations
  - The *auipc+jr/jalr* sequence is used because the offset from the PC is out of the ±1MB range.

If a return address stack is implemented, then as *cm.jalt* is equivalent to *jal ra*, it pushes to the stack.

4.1. JVT

The base of the table is in the JVT CSR (see JVT CSR, table jump base vector and control register), each table entry is XLEN bits.

If the same function is called with and without linking then it must have two entries in the table. This is typically caused by the same function being called with and without tail calling.

4.2. Table Jump Fault handling

For a table jump instruction, the table entry that the instruction selects is considered an extension of the instruction itself. Hence, the execution of a table jump instruction involves two instruction fetches, the first to read the instruction (*cm.jt*/*cm.jalt*) and the second to read from the jump vector table (JVT). Both instruction fetches are *implicit* reads, and both require execute permission; read permission is irrelevant. It is recommended that the second fetch be ignored for hardware triggers and breakpoints.

Memory writes to the jump vector table require an instruction barrier (*fence.i*) to guarantee that they are visible to the instruction fetch.

Multiple contexts may have different jump vector tables. JVT may be switched between them without an instruction barrier if the tables have not been updated in memory since the last *fence.i*.

If an exception occurs on either instruction fetch, xEPC is set to the PC of the table jump instruction, xCAUSE is set as expected for the type of fault and xTVAL (if not set to zero) contains the fetch address which caused the fault.

Included in

Extension	Minimum version	Lifecycle state
Zcmt (Zcmt)	v1.0	Ratified

4.3. JVT CSR

Synopsis

Table jump base vector and control register

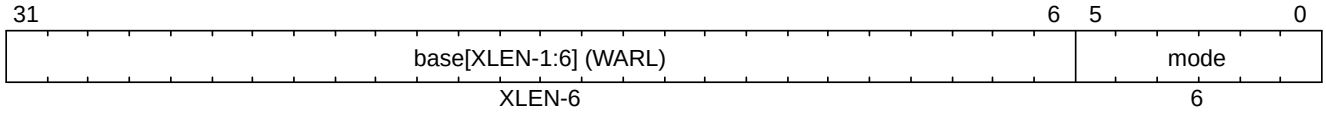
Address

0x0017

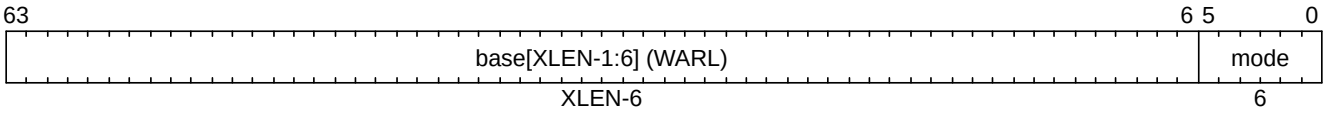
Permissions

URW

Format (RV32)



Format (RV64)



Description

The *JVT* register is an XLEN-bit **WARL** read/write register that holds the jump table configuration, consisting of the jump table base address (BASE) and the jump table mode (MODE).

If Zcmt is implemented then *JVT* must also be implemented, but can contain a read-only value. If *JVT* is writable, the set of values the register may hold can vary by implementation. The value in the BASE field must always be aligned on a 64-byte boundary.

JVT.base is a virtual address, whenever virtual memory is enabled.

The memory pointed to by *JVT.base* is treated as instruction memory for the purpose of executing table jump instructions, implying execute access permission.

Table 3. JVT.mode definition

JVT.mode	Comment
000000	Jump table mode
others	**reserved for future standard use**

JVT.mode is a **WARL** field, so can only be programmed to modes which are implemented. Therefore the discovery mechanism is to attempt to program different modes and read back the values to see which are available. Jump table mode *must* be implemented.

NOTE in future the RISC-V Unified Discovery method will report the available modes.

Architectural State

JVT adds architectural state to the system software context (such as an OS process), therefore must be saved/restored on context switches.

State Enable

If the Smstateen extension is implemented, then bit 2 in *mstateen0*, *sstateen0*, and *hstateen0* is implemented. If bit 2 of a controlling *stateen0* CSR is zero, then access to the *JVT* CSR and execution of a *cm.jalt* or *cm.jt* instruction by a lower privilege level results in an Illegal Instruction trap (or, if appropriate, a Virtual Instruction trap).

Included in

Extension	Minimum version	Lifecycle state
Zcmt (Zcmt)	v1.0	Ratified

4.4. cm.jt

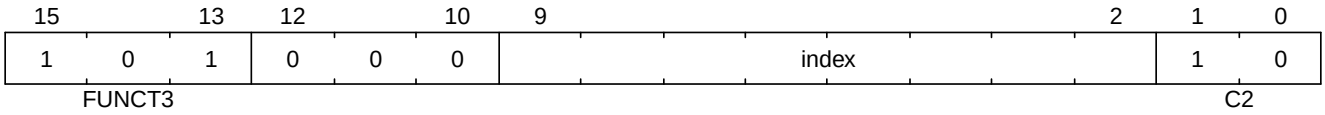
Synopsis

jump via table

Mnemonic

cm.jt *index*

Encoding (RV32, RV64)



NOTE For this encoding to decode as *cm.jt*, *index<32*, otherwise it decodes as *cm.jalt*, see Jump and link via table.

NOTE If JVT.mode = 0 (Jump Table Mode) then *cm.jt* behaves as specified here. If JVT.mode is a reserved value, then *cm.jt* is also reserved. In the future other defined values of JVT.mode may change the behaviour of *cm.jt*.

Assembly Syntax

```
cm.jt index
```

Description

cm.jt reads an entry from the jump vector table in memory and jumps to the address that was read.

For further information see Table Jump Overview.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists.

Operation

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

# target_address is temporary internal state, it doesn't represent a real register
# InstMemory is byte indexed

switch(XLEN) {
  32:  table_address[XLEN-1:0] = JVT.base + (index<<2);
  64:  table_address[XLEN-1:0] = JVT.base + (index<<3);
}

//fetch from the jump table
target_address[XLEN-1:0] = InstMemory[table_address][XLEN-1:0];

j target_address[XLEN-1:0]&~0x1;
```

Included in

Extension	Minimum version	Lifecycle state
Zcmt (Zcmt)	v1.0	Ratified

4.5. cm.jalt

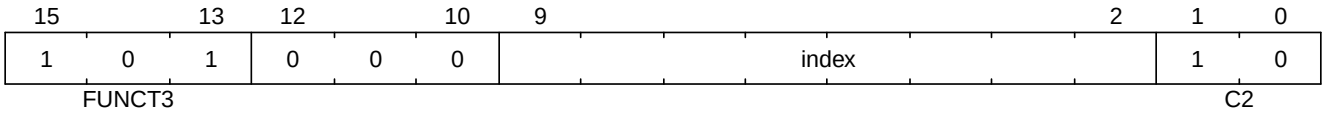
Synopsis

jump via table with optional link

Mnemonic

cm.jalt *index*

Encoding (RV32, RV64)



NOTE For this encoding to decode as *cm.jalt*, *index>=32*, otherwise it decodes as *cm.jt*, see Jump via table.

NOTE If JVT.mode = 0 (Jump Table Mode) then *cm.jalt* behaves as specified here. If JVT.mode is a reserved value, then *cm.jalt* is also reserved. In the future other defined values of JVT.mode may change the behaviour of *cm.jalt*.

Assembly Syntax

```
cm.jalt index
```

Description

cm.jalt reads an entry from the jump vector table in memory and jumps to the address that was read, linking to *ra*.

For further information see Table Jump Overview.

Prerequisites

None

32-bit equivalent

No direct equivalent encoding exists.

Operation

```
//This is not SAIL, it's pseudo-code. The SAIL hasn't been written yet.

# target_address is temporary internal state, it doesn't represent a real register
# InstMemory is byte indexed

switch(XLEN) {
  32:  table_address[XLEN-1:0] = JVT.base + (index<<2);
  64:  table_address[XLEN-1:0] = JVT.base + (index<<3);
}

//fetch from the jump table
target_address[XLEN-1:0] = InstMemory[table_address][XLEN-1:0];

jal ra, target_address[XLEN-1:0]&~0x1;
```

Included in

Extension	Minimum version	Lifecycle state
Zcmt (Zcmt)	v1.0	Ratified