

Zc* v0.70.3

Change history since v0.70.1 (tagged release)

Table 1. Change history

Version	change
v0.70.3	added rule that Zcf implies Zca (this text was missing, this is not a spec change), and added push/pop examples in the push/pop section
v0.70.2	stylistic changes only, removing redundant text. Also corrected field names on JVT CSR diagram, and fixed synopsis for cm.mvsa01

Zc* Overview

This document is in the Stable state. Assume anything could still change, but limited change should be expected. For more information see: <https://riscv.org/spec-state>

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

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

Table 2. Zc* extension overview

Instruction	Zca	Zcf	Zcb	Zcmb	Zcmp	Zcmpe	Zcmt
Define a subset of C with the FP load/stores removed							
C excl. c.f*	✓						
The single precision floating point load/stores become a separate extension							
c.flw		✓					
c.flwsp		✓					
c.fsw		✓					
c.fswsp		✓					
Simple operations for use on all architectures							
c.lbu			✓				
c.lh			✓				
c.lhu			✓				
c.sb			✓				
c.sh			✓				
c.zext.b			✓				
c.sext.b			✓				

Instruction	Zca	Zcf	Zcb	Zcmb	Zcmp	Zcmpe	Zcmt
c.zext.h			✓				
c.sext.h			✓				
c.zext.w			✓				
c.mul			✓				
c.not			✓				
Load/store byte/half which overlap with <i>c.fld</i> , <i>c.fldsp</i> , <i>c.fsd</i>							
cm.lb				✓			
cm.lbu				✓			
cm.lh				✓			
cm.lhu				✓			
cm.sb				✓			
cm.sh				✓			
PUSH/POP and double move which overlap with <i>c.fsdsp</i>							
cm.push					✓	✓	
cm.pop					✓	✓	
cm.popret					✓	✓	
cm.popretz					✓	✓	
cm.mva0ls					✓		
cm.mvsa0l					✓		
Reserved for EABI versions of PUSH/POP and double move which overlap with <i>c.fsdsp</i>							
cm.push.e						✓	
cm.pop.e						✓	
cm.popret.e						✓	
cm.popretz.e						✓	
cm.mva0ls.e						✓	
cm.mvsa0l.e						✓	
Table jump							
cm.jt							✓
cm.jalt							✓

Zca

Zca is all of the existing C extension, *excluding* all 16-bit floating point loads and stores: *c.flw*, *c.flwsp*,

c.fsw, c.fswsp, c.fld, c.fldsp, c.fsd, c.fsdsp.

Zcf

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

Zcb requires the [Zca](#) extension.

Zcb

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

Zcb requires the [Zca](#) extension.

The *c.mul* encoding uses the CR 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
✓	✓	<i>c.lbu rd', uimm(rs1')</i>	c.lbu : Load unsigned byte, 16-bit encoding
✓	✓	<i>c.lhu rd', uimm(rs1')</i>	c.lhu : Load unsigned halfword, 16-bit encoding
✓	✓	<i>c.lh rd', uimm(rs1')</i>	c.lh : Load signed halfword, 16-bit encoding
✓	✓	<i>c.sb rs2', uimm(rs1')</i>	c.sb : Store byte, 16-bit encoding
✓	✓	<i>c.sh rs2', uimm(rs1')</i>	c.sh : Store halfword, 16-bit encoding
✓	✓	<i>c.zext.b rsd'</i>	c.zext.b : Zero extend byte, 16-bit encoding
✓	✓	<i>c.sext.b rsd'</i>	c.sext.b : Sign extend byte, 16-bit encoding
✓	✓	<i>c.zext.h rsd'</i>	c.zext.h : Zero extend halfword, 16-bit encoding
✓	✓	<i>c.sext.h rsd'</i>	c.sext.h : Sign extend halfword, 16-bit encoding
	✓	<i>c.zext.w rsd'</i>	c.zext.w : Zero extend word, 16-bit encoding
✓	✓	<i>c.not rsd'</i>	c.not : Bitwise not, 16-bit encoding
✓	✓	<i>c.mul rsd', rs2'</i>	c.mul : Multiply, 16-bit encoding

Zcmb

This extension reuses some encodings from *c.fld*, *c.fldsp*, and *c.fsd*. Therefore it is *incompatible* with the full C-extension. It is compatible with F, D with Zdinx.

Zcmb requires the [Zcb](#) extension, which in turn requires the [Zca](#) extension.

The instructions are all 16-bit versions of existing 32-bit load/store instructions.

RV32	RV64	Mnemonic	Instruction
✓	✓	cm.lbu <i>rd'</i> , uimm(<i>rs1'</i>)	cm.lbu : Load unsigned byte, 16-bit encoding
✓	✓	cm.lhu <i>rd'</i> , uimm(<i>rs1'</i>)	cm.lhu : Load unsigned halfword, 16-bit encoding
✓	✓	cm.lb <i>rd'</i> , uimm(<i>rs1'</i>)	cm.lb : Load signed byte, 16-bit encoding
✓	✓	cm.lh <i>rd'</i> , uimm(<i>rs1'</i>)	cm.lh : Load signed halfword, 16-bit encoding
✓	✓	cm.sb <i>rs2'</i> , uimm(<i>rs1'</i>)	cm.sb : Store byte, 16-bit encoding
✓	✓	cm.sh <i>rs2'</i> , uimm(<i>rs1'</i>)	cm.sh : Store halfword, 16-bit encoding

Zcmp

Zcmp is the set of sequenced instructions for code-size reduction.

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with the full C-extension. It is compatible with F, D with Zdinx.

Zcmp requires 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
✓	✓	cm.push { <i>reg_list</i> }, - <i>stack_adj</i>	cm.push: Create stack frame: push registers, allocate additional stack space.
✓	✓	cm.pop { <i>reg_list</i> }, <i>stack_adj</i>	cm.pop: Destroy stack frame: pop registers, deallocate stack frame.
✓	✓	cm.popret { <i>reg_list</i> }, <i>stack_adj</i>	cm.popret: Destroy stack frame: pop registers, deallocate stack frame, return.
✓	✓	cm.popretz { <i>reg_list</i> }, <i>stack_adj</i>	cm.popretz: Destroy stack frame: pop registers, deallocate stack frame, return zero.
✓	✓	cm.mva01s <i>sreg1</i> , <i>sreg2</i>	cm.mva01s: move two s0-s7 registers into a0-a1
✓	✓	cm.mvsa01 <i>sreg1</i> , <i>sreg2</i>	cm.mvsa01: move a0-a1 into two different s0-s7 registers

Zcmpe

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with the full C-extension. It is compatible with F, D with Zdinx.

Zcmpe requires the [Zca](#) extension.

Zcmpe offers EABI support for register mappings from [Zcmp](#) where the *x* register mapping is different to the UABI. The EABI specification is not frozen so these instructions cannot yet be accurately specified.

Zcmt

Zcmt is the set of table jump instructions for code-size reduction.

This extension reuses some encodings from *c.fsdsp*. Therefore it is *incompatible* with the full C-extension. It is compatible with F, D with Zdinx.

RV32	RV64	Mnemonic	Instruction
✓	✓	cm.jt <i>index</i>	cm.jt : jump via table without link
✓	✓	cm.jalt <i>index</i>	cm.jalt : jump via table and link to ra

c.lbu

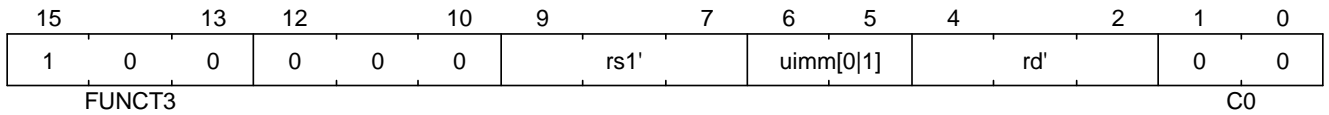
Synopsis

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.*
- NOTE***For an longer immediate with a 16-bit encoding see [cm.lbu: Load unsigned byte, 16-bit encoding](#).*
- NOTE***To load signed bytes with a 16-bit encoding see [cm.lb: Load signed byte, 16-bit encoding](#).*

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)	v0.70.3	Stable

c.lhu

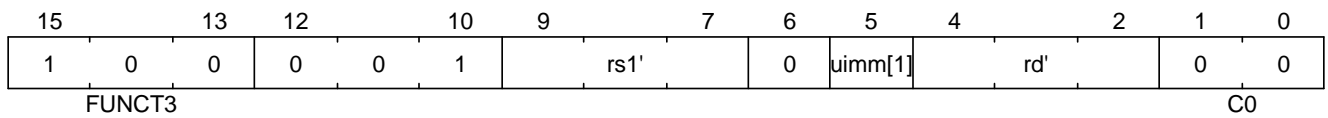
Synopsis

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.*
- NOTE***For an longer immediate with a 16-bit encoding see [cm.lhu: Load unsigned halfword, 16-bit encoding](#).*

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)	v0.70.3	Stable

c.lh

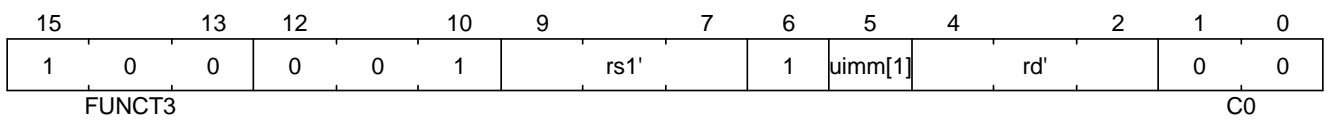
Synopsis

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.*
- NOTE***For an longer immediate with a 16-bit encoding see [cm.lh: Load signed halfword, 16-bit encoding](#).*

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)	v0.70.3	Stable

c.sb

Synopsis

Store byte, 16-bit encoding

Mnemonic

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

Encoding (RV32, RV64)

15	13	12	10	9	7	6	5	4	2	1	0
1	0	0	0	1	0	rs1'		uimm[0 1]	rs2'		0 0
FUNCT3										C0	

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.*

NOTE *For an longer immediate with a 16-bit encoding see [cm.sb: Store byte, 16-bit encoding](#).*

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)	v0.70.3	Stable

c.sh

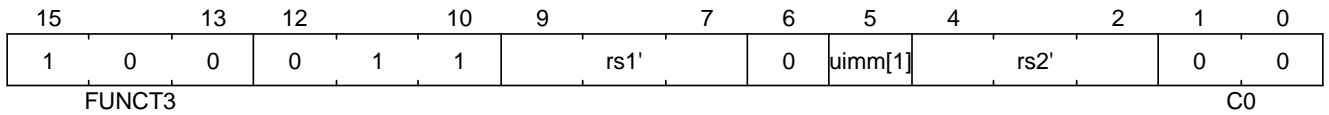
Synopsis

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*.
- NOTE** | For an longer immediate with a 16-bit encoding see [cm.sh: Store halfword, 16-bit encoding](#).

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)	v0.70.3	Stable

c.zext.b

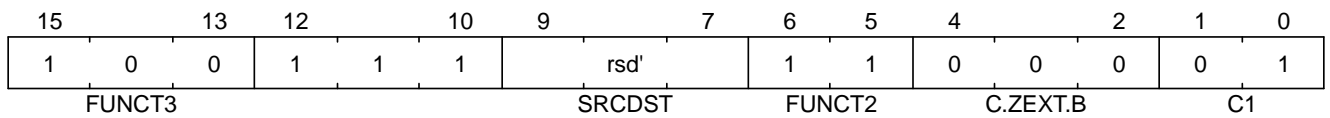
Synopsis

Zero extend byte, 16-bit encoding

Mnemonic

c.zext.b *rsd'*

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 | *rsd' is from the standard 8-register set x8-x15.*

Prerequisites

None

32-bit equivalent

```
andi rsd, rsd, 0xff
```

NOTE | *The SAIL module variable for rsd' is called rsdc.*

Operation

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

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v0.70.3	Stable

c.sext.b

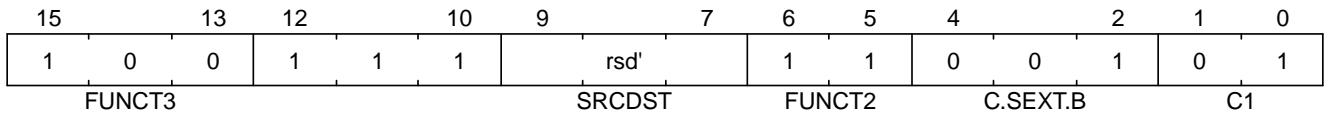
Synopsis

Sign extend byte, 16-bit encoding

Mnemonic

c.sext.b *rsd'*

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

rsd' is from the standard 8-register set x8-x15.

Prerequisites

Zbb must also be configured.

32-bit equivalent

[\[insns-sext_b\]](#) from Zbb

NOTE

The SAIL module variable for rsd' is called rsdc.

Operation

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

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v0.70.3	Stable

c.zext.h

Synopsis

Zero extend halfword, 16-bit encoding

Mnemonic

c.zext.h *rsd'*

Encoding (RV32, RV64)

15	13	12	10	9	7	6	5	4	2	1	0			
1	0	0	1	1	1	rsd'		1	1	0	1	0	0	1
FUNCT3			SRCDST			FUNCT2		C.ZEXT.H			C1			

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 *rsd' is from the standard 8-register set x8-x15.*

Prerequisites

Zbb must also be configured.

32-bit equivalent

[\[insns-zext_h\]](#) from Zbb

NOTE *The SAIL module variable for rsd' is called rsdc.*

Operation

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

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v0.70.3	Stable

c.sext.h

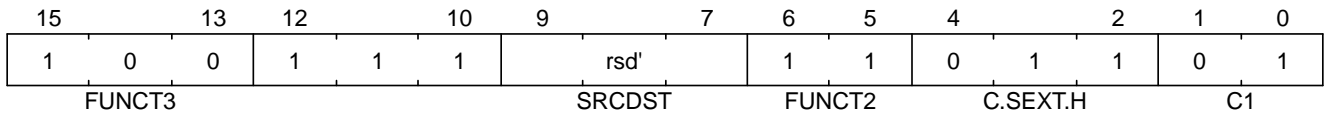
Synopsis

Sign extend halfword, 16-bit encoding

Mnemonic

c.sext.h *rsd'*

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

rsd' is from the standard 8-register set x8-x15.

Prerequisites

Zbb must also be configured.

32-bit equivalent

[\[insns-sext_h\]](#) from Zbb

NOTE

*The SAIL module variable for *rsd'* is called *rsdc*.*

Operation

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

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v0.70.3	Stable

c.zext.w

Synopsis

Zero extend word, 16-bit encoding

Mnemonic

c.zext.w *rsd'*

Encoding (RV64)

15	13	12	10	9	7	6	5	4	2	1	0			
1	0	0	1	1	1	rsd'		1	1	1	0	0	0	1
FUNCT3			SRCDST			FUNCT2		C.ZEXT.W			C1			

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 | *rsd' is from the standard 8-register set x8-x15.*

Prerequisites

Zba must also be configured.

32-bit equivalent

```
add.uw rsd', rsd', zero
```

NOTE | *The SAIL module variable for rsd' is called rsdc.*

Operation

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

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v0.70.3	Stable

c.not

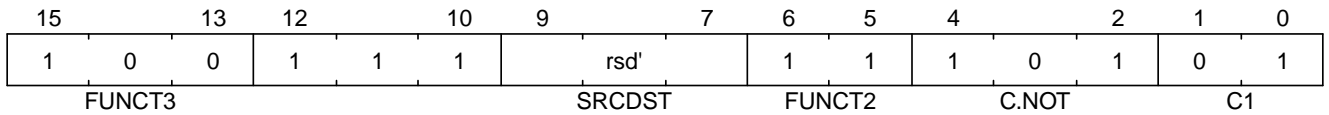
Synopsis

Bitwise not, 16-bit encoding

Mnemonic

c.not *rsd'*

Encoding (RV32, RV64)



Description

This instruction takes the one’s complement of *rsd'* and writes the result to the same register.

NOTE

rsd' is from the standard 8-register set x8-x15.

Prerequisites

None

32-bit equivalent

```
xori rd, rs, -1
```

NOTE

The SAIL module variable for *rsd'* is called *rsdc*.

Operation

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

Included in

Extension	Minimum version	Lifecycle state
Zcb (Zcb)	v0.70.3	Stable

c.mul

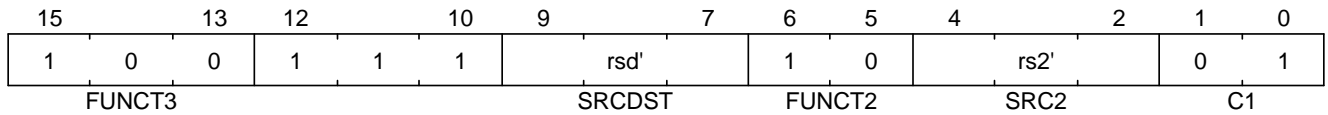
Synopsis

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

rsd' 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 rsd' 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)	v0.70.3	Stable

cm.lbu

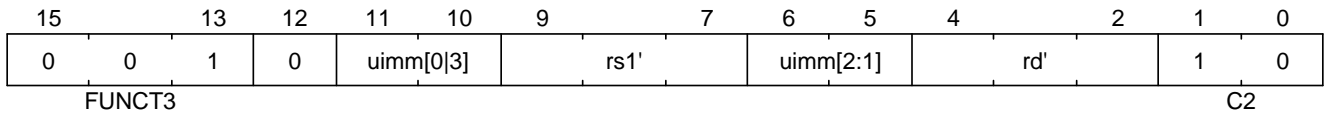
Synopsis

Load unsigned byte, 16-bit encoding

Mnemonic

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

Encoding (RV32, RV64)



NOTE

If *uimm* < 4 the encoding is designated for custom use, as the functionality overlaps with [c.lbu](#): Load unsigned byte, 16-bit encoding.

The immediate offset is formed as follows:

```
uimm[31:4] = 0;
uimm[3]     = encoding[10];
uimm[2:1]   = encoding[6:5];
uimm[0]     = encoding[11];
```

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
Zcmb (Zcmb)	v0.70.3	Stable

cm.lhu

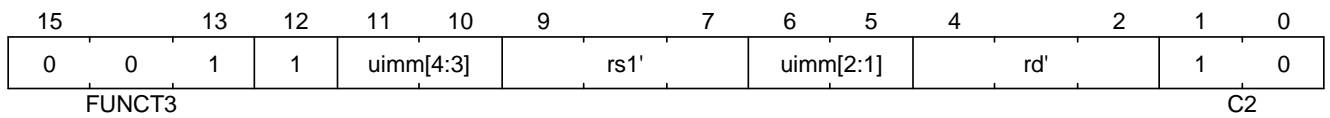
Synopsis

Load unsigned halfword, 16-bit encoding

Mnemonic

cm.lhu rd', uimm(rs1')

Encoding (RV32, RV64)



NOTE

If *uimm* < 4 the encoding is designated for custom use, as the functionality overlaps with [c.lhu: Load unsigned halfword, 16-bit encoding](#).

The immediate offset is formed as follows:

```
uimm[31:5] = 0;
uimm[4:3]   = encoding[11:10];
uimm[2:1]   = encoding[6:5];
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
Zcmb (Zcmb)	v0.70.3	Stable

cm.lb

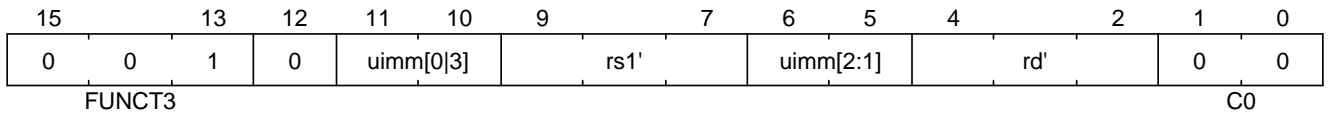
Synopsis

Load signed byte, 16-bit encoding

Mnemonic

cm.lb rd', uimm(rs1')

Encoding (RV32, RV64)



The immediate offset is formed as follows:

```
uimm[31:4] = 0;
uimm[3]     = encoding[10];
uimm[2:1]   = encoding[6:5];
uimm[0]     = encoding[11];
```

Description

This instruction loads a byte from the memory address formed by adding *rs1'* to the zero extended immediate *uimm*. The resulting byte 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

[\[insns-lb\]](#)

Operation

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

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

Included in

Extension	Minimum version	Lifecycle state
Zcmb (Zcmb)	v0.70.3	Stable

cm.lh

Synopsis

Load signed halfword, 16-bit encoding

Mnemonic

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

Encoding (RV32, RV64)

15	13	12	11	10	9	7	6	5	4	2	1	0
0	0	1	1	uimm[4:3]		rs1'		uimm[2:1]		rd'	0	0
FUNCT3											C0	

NOTE

If *uimm* < 4 the encoding is designated for custom use, as the functionality overlaps with [c.lh: Load signed halfword, 16-bit encoding](#).

The immediate offset is formed as follows:

```
uimm[31:5] = 0;
uimm[4:3]  = encoding[11:10];
uimm[2:1]  = encoding[6:5];
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
Zcmb (Zcmb)	v0.70.3	Stable

cm.sb

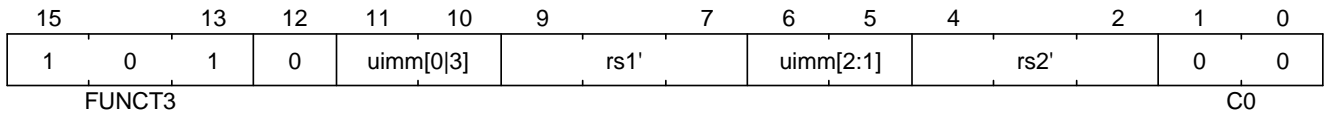
Synopsis

Store byte, 16-bit encoding

Mnemonic

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

Encoding (RV32, RV64)



NOTE

If *uimm* < 4 the encoding is designated for custom use, as the functionality overlaps with [c.sb: Store byte, 16-bit encoding](#).

The immediate offset is formed as follows:

```
uimm[31:4] = 0;
uimm[3]    = encoding[10];
uimm[2:1]  = encoding[6:5];
uimm[0]    = encoding[11];
```

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
Zcmb (Zcmb)	v0.70.3	Stable

cm.sh

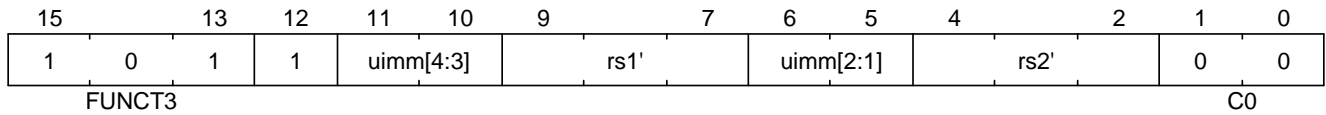
Synopsis

Store halfword, 16-bit encoding

Mnemonic

cm.sh $rs2'$, $uimm(rs1')$

Encoding (RV32, RV64)



NOTE

If $uimm < 4$ the encoding is designated for custom use, as the functionality overlaps with [c.sh: Store halfword, 16-bit encoding](#).

The immediate offset is formed as follows:

```
uimm[31:5] = 0;
uimm[4:3]  = encoding[11:10];
uimm[2:1]  = encoding[6:5];
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
Zcmb (Zcmb)	v0.70.3	Stable

PUSH/POP register instructions

These instructions are collectively referred to as PUSH/POP:

- `cm.push`: Create stack frame: push registers, allocate additional stack space.
- `cm.pop`: Destroy stack frame: pop registers, deallocate stack frame.
- `cm.popret`: Destroy stack frame: pop registers, deallocate stack frame, return.
- `cm.popretz`: Destroy stack frame: 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.

PUSH/POP functional overview

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

1. The PUSH instruction
 - pushes (stores) the registers specified in the register list to the stack frame
 - adjusts the stack pointer to create 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 from
 - `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

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 $\pm 1\text{MB}$ 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.

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 by extra 16-byte blocks. 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.

Register list handling

The instructions do not directly support $\{ra, s0-s10\}$ to reduce the amount of encoding space required. If this register list is required then *s11* should also be included. This costs a small amount of memory and performance, but saves code-size.

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 faults detected during the sequence. The entire PUSH/POP sequence is re-executed after returning from the fault handler, and multiple faults are possible during the sequence.

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

Software view of execution

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

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 complete without triggering any precise faults (for example, page faults), and without the core taking an interrupt.

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
```

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 an implementation allows interrupts during the sequence, then any loads which were executed before the interrupt may update architectural state. The loads will be re-executed once the handler completes, so the values will be overwritten. Therefore it is permitted for an implementation to update some of the destination registers before taking an interrupt or other 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 complete without triggering any precise faults (for example, page faults), and without the core taking an interrupt.

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
```

Forward progress guarantee

The PUSH/POP sequence has the same forward progress guarantee as executing the instructions from the equivalent assembly sequences.

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.

If the core implementation does support PUSH/POP to non-idempotent memory, then it may not be possible to re-execute the sequence after a fault. In this case the fault handler should complete the sequence in software.

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.

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

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

expands to:

```
sw  s2,  -4(sp);
sw  s1,  -8(sp);
sw  s0, -16(sp);
sw  ra, -20(sp);
addi sp, sp, -64;
```

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;
```


cm.pop {ra}, 16

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

expands to:

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

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;
```

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)	v0.70.3	Stable

cm.push

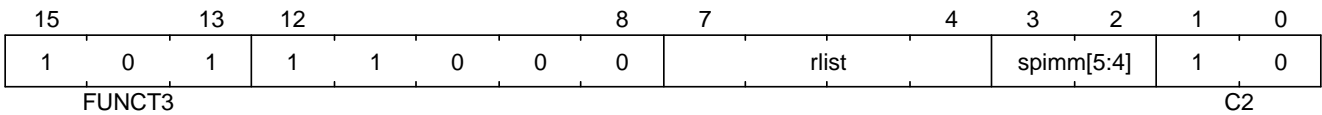
Synopsis

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: take_illegal_instruction_exception();
}

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: take_illegal_instruction_exception();
}
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 blocks, required to cover the registers in the list.

spimm is the number of additional 16-byte blocks 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)	v0.70.3	Stable

Included in

Extension	Minimum version	Lifecycle state
Zcmpe (Zcmpe)	v0.70.3	Stable

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_adj

Encoding (RV32, RV64)

15	13	12	8	7	4	3	2	1	0		
1	0	1	1	1	0	1	0	rlist	spimm[5:4]	1	0
FUNCT3								C2			

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: take_illegal_instruction_exception();
}
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: take_illegal_instruction_exception();
}
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 blocks, required to cover the registers in the list.

spimm is the number of additional 16-byte blocks 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)	v0.70.3	Stable

Included in

Extension	Minimum version	Lifecycle state
Zcmpe (Zcmpe)	v0.70.3	Stable

cm.popretz

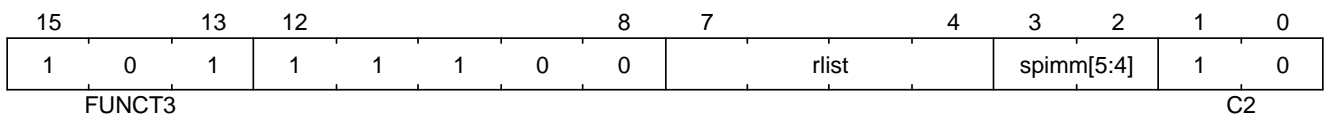
Synopsis

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: take_illegal_instruction_exception();
}
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: take_illegal_instruction_exception();
}
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 blocks, required to cover the registers in the list.

spimm is the number of additional 16-byte blocks 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)	v0.70.3	Stable

Included in

Extension	Minimum version	Lifecycle state
Zcmpe (Zcmpe)	v0.70.3	Stable

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)

15	13	12	8	7	4	3	2	1	0
1	0	1	1	1	1	0	rlist	spimm[5:4]	1 0
FUNCT3								C2	

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: take_illegal_instruction_exception();
}
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: take_illegal_instruction_exception();
}
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 blocks, required to cover the registers in the list.

spimm is the number of additional 16-byte blocks 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)	v0.70.3	Stable

Included in

Extension	Minimum version	Lifecycle state
Zcmpe (Zcmpe)	v0.70.3	Stable

cm.mvsa01

Synopsis

Move a0-a1 into two registers of s0-s7

Mnemonic

cm.mvsa01 sreg1, sreg2

Encoding (RV32, RV64)

15	13	12	10	9	7	6	5	4	2	1	0		
1	0	1	0	1	1	sreg1		0	1	sreg2		1	0
FUNCT3										C2			

NOTE For the encoding to be legal sreg1 != sreg2.

Assembly Syntax

```
cm.mvsa01 sreg1, sreg2
```

Description

This instruction moves a0 into sreg1 and a1 into sreg2. sreg1 and sreg2 must be different. The execution is atomic, so it is not possible to observe state where only one of sreg1 or sreg2 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 && (sreg1>1 || sreg2>1)) {
    take_illegal_instruction_exception();
}
xreg1 = {sreg1[2:1]>0,sreg1[2:1]==0,sreg1[2:0]};
xreg2 = {sreg2[2:1]>0,sreg2[2:1]==0,sreg2[2:0]};
X[xreg1] = X[10];
X[xreg2] = X[11];
```

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v0.70.3	Stable

cm.mva01s

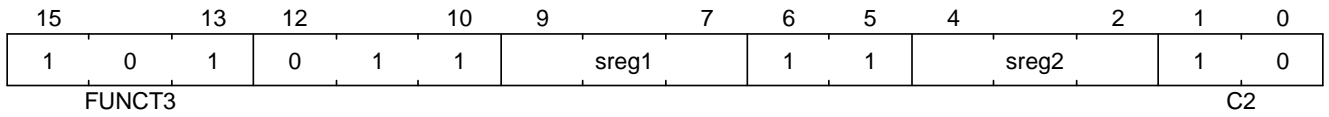
Synopsis

Move two s0-s7 registers into a0-a1

Mnemonic

cm.mva01s *sreg1*, *sreg2*

Encoding (RV32, RV64)



Assembly Syntax

```
cm.mva01s sreg1, sreg2
```

Description

This instruction moves *sreg1* into *a0* and *sreg2* 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 && (sreg1>1 || sreg2>1)) {
    take_illegal_instruction_exception();
}
xreg1 = {sreg1[2:1]>0,sreg1[2:1]==0,sreg1[2:0]};
xreg2 = {sreg2[2:1]>0,sreg2[2:1]==0,sreg2[2:0]};
X[10] = X[xreg1];
X[11] = X[xreg2];
```

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v0.70.3	Stable

Table Jump Instructions

These instructions are collectively referred to as table jump:

- [cm.jt](#): jump via table without link
- [cm.jalt](#): jump via table and link to ra

Common details for these instructions are in this section.

Table Jump Overview

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

Function calls and jumps to fixed labels typically take 32-bit or 64-bit instruction sequences.

Table jump allows the linker to:

- replace 32-bit *j* calls with *cm.jt*
- replace 32-bit *jal* ra calls with *cm.jalt*
- replace 64-bit *auipc/jalr* calls to fixed locations with *cm.jt*
- replace 64-bit *auipc/jalr* ra calls to fixed locations with *cm.jalt*
 - The *auipc+jr/jalr* sequence is used because the offset from the PC is out of the $\pm 1\text{MB}$ range.

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.

The table entry number is from the *index* field in the encoding, which controls the link register.

- *cm.jt* : entries 0-63, link to zero
- *cm.jalt* : entries 64-255, link to ra

Note that the LSB of every jump vector table entry is *ignored* which matches standard *jalr* behaviour.

If the same function is called with and without linking then it must have two entries in the table. This case does happen in practice but only affects a small number of entries so it does not waste much space in the table. It is typically caused by the same function being called with and without tail calling.

Recommended algorithm for allocating entries in the jump vector table

Calls to each function are categorised as shown in [Table jump code size saving for each function call replacement](#).

Table 3. Table jump code size saving for each function call replacement

original sequence	Table Jump saving
<i>j</i>	$A*2-(XLEN/8)$ bytes
<i>auipc+jr</i>	$B*6-(XLEN/8)$ bytes
<i>jal ra</i>	$C*2-(XLEN/8)$ bytes
<i>auipc+jalr ra</i>	$D*6-(XLEN/8)$ bytes

Each function is called by using one of the two link registers. The total saving per function is calculated by counting the number of calls and adding up the total saving from each replacement of the existing sequence with a Table Jump instruction, as follows:

```
saving_per_function_cm_jt      = A * 2 + B * 6 - 2*(XLEN-8)
saving_per_function_cm_jalt    = C * 2 + D * 6 - 2*(XLEN-8)
```

The functions are sorted so that the one with the highest saving is in table entry 0, the second highest in entry 1 etc. for that encoding.

NOTE

This algorithm assumes that each function is only called with one link register. If the same function is called with more than one link register, then it must have two entries in the table.

This allows the core to cache the most frequent targets by caching the lowest numbered entries of each section of the jump vector table. Only caching a few entries will greatly improve the performance.

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 main instruction (*cm.jt* or *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.

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 address which caused the fault.

Included in

Extension	Minimum version	Lifecycle state
Zcmp (Zcmp)	v0.70.3	Stable

JVT CSR

Synopsis

Table jump base vector and control register

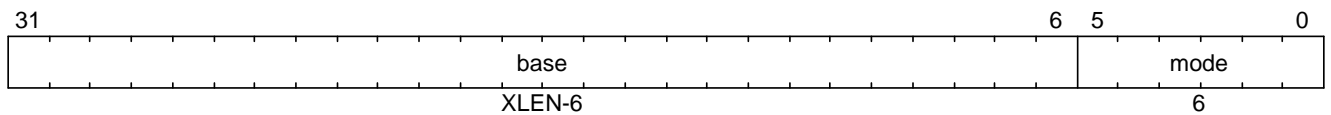
Address

0x0017

Permissions

URW

Format (RV32, RV64)



Description

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

JVT.base[5:0] is implicitly zero, and is naturally aligned for all legal values of *XLEN*.

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

Table 4. *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.

Architectural State

JVT adds architectural state to the context, therefore must be saved/restored on context switches.

Additional architectural state requires a state enable to be allocated. Accesses when the state is disabled will throw an illegal instruction exception. The state enable is not specified in this document.

Included in

Extension	Minimum version	Lifecycle state
Zcmt (Zcmt)	v0.70.3	Stable

cm.jt

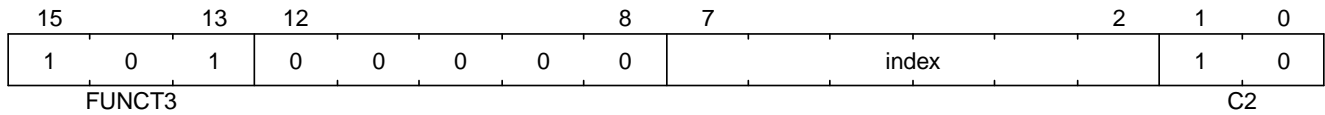
Synopsis

jump via table without link

Mnemonic

cm.jt *index*

Encoding (RV32, RV64)



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, without linking.

For further information see [Table Jump Instructions](#).

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];

//jump to the target address
jr target_address[XLEN-1:0]&~0x1;
```

Included in

Extension	Minimum version	Lifecycle state
Zcmt (Zcmt)	v0.70.3	Stable

cm.jalt

Synopsis

jump via table and link to ra

Mnemonic

cm.jalt *index*

Encoding (RV32, RV64)

15	13	12	10	9		2	1	0
1	0	1	0	0	0	index	1	0
FUNCT3						C2		

NOTE

For this encoding to decode as *cm.jalt*, $index \geq 64$, otherwise it decodes as *cm.jt: jump via table without link*.

NOTE

The equivalent encoding with $bit[10]=1$ is reserved to allow future expansion of the table index.

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 Instructions](#).

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];

//jump to the target address
jalr ra, target_address[XLEN-1:0]&~0x1;
```

Included in

Extension	Minimum version	Lifecycle state
Zcmt (Zcmt)	v0.70.3	Stable