Zces 0.53.1

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

Zces is the set of sequenced or more complex instuctions for code-size reduction.

This extension reuses encodings from the D-extension. Therefore it is *incompatible* with D. It is fully compatible with F and also with Zdinx.

jt and jalt require JVT CSR, table jump base vector and control register.

RV 32	RV 64	Mnemonic	Instruction
✓	✓	c.push {reg_list_1_4}, {areg_list}, -stack_adj	c.pusha: push 2 to 4 registers to stack memory and move argument registers into saved registers, 16-bit encoding
✓	✓	c.push {reg_list_5_13}, {areg_list}, -stack_adj	c.pusha.l: push 5 to 13 registers to stack memory and move argument registers into saved registers, 16-bit encoding
✓	✓	c.push {reg_list_1_4}, -stack_adj	c.push: push 1 to 4 registers to stack memory, 16-bit encoding
✓	✓	c.push {reg_list_5_13}, -stack_adj	c.push.l: push 5 to 13 registers to stack memory, 16-bit encoding
✓	✓	c.pop {reg_list_1_4}, stack_adj	c.pop: pop 1 to 4 registers from stack memory, 16-bit encoding
✓	✓	c.popret {reg_list_1_4}, stack_adj	c.popret: pop 1 to 4 registers from stack memory and return, 16-bit encoding
✓	✓	c.popret {reg_list_5_13}, stack_adj	c.popret.l: pop 5 to 13 registers from stack memory and return, 16-bit encoding
✓	✓	c.popretz {reg_list_1_4}, stack_adj	c.popretz: pop 1 to 4 registers from stack memory and return zero, 16-bit encoding
✓	✓	c.popretz {reg_list_5_13}, stack_adj	c.popretz.l: pop 5 to 13 registers from stack memory and returns zero, 16-bit encoding
✓	✓	c.jt #index	c.jt: jump via table without link, 16-bit encoding
✓	✓	c.jalt #index	c.jalt: jump via table and link to ra, 16-bit encoding
✓	✓	c.mva01s sreg1, sreg2	c.mva01s: move two s0-s7 registers into a0-a1, 16-bit encoding

c.pusha

Synopsis

Push 2 to 4 registers to stack memory and move argument registers into saved registers, 16-bit encoding

Mnemonic

c.push *{reg_list_2_4}*, *{areg_list}*, *-stack_adj*

Encoding (RV32, RV64)

15					10	9	8 7	6		4	3 2	1	0
1	0	1	0	1	0	0	spimm[5:4]	0	1	1	rlist2	1	0
FUNCT6											reg_list_2_4	OP=C2	

Syntax

The variables used in the syntax are defined below.

```
<reg_list_2_4> ::= <ra> "," [<s0>  | <s0-sN>] (where N is 1,2)

if (<reg_list_2_4>=="ra, s0")       <xreg_list_2_4>="x1, x8"
if (<reg_list_2_4>=="ra, s0-s1")    <xreg_list_2_4>="x1, x8-x9"
if (<reg_list_2_4>=="ra, s0-s2")    <xreg_list_2_4>="x1, x8-x9, x18"

if (<reg_list_2_4>=="ra, s0")       <areg_list>="a0"
if (<reg_list_2_4>=="ra, s0-s1")    <areg_list>="a0-a1"
if (<reg_list_2_4>=="ra, s0-s2")    <areg_list>="a0-a2"

RV32:
<stack_adj>=[16|32|48|64]

RV64:
if (<reg_list_2_4>=="ra, s0")       <stack_adj>=[16|32|48|64|96]
if (<reg_list_2_4>=="ra, s0-s1")    <stack_adj>=[32|48|64|96|112]
if (<reg_list_2_4>=="ra, s0-s2")    <stack_adj>=[32|48|64|96|112]
```

Description

This instruction pushes (stores) the registers in *reg_list_2_4* to stack memory, moves *areg_list* into similarly numbered *s* registers, and then adjusts the stack pointer by -*stack_adj*.

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

For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist2* field in the encoding are as shown below.

Table 1. rlist2 decoding

rlist2	**reg_list_2_4**	**stack_adj_base**	
		RV32	RV64
0	reserved encoding		
1	ra, s0	16	16
2	ra, s0-s1	16	32
3	ra, s0-s2	16	32

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_2_4*.

spimm is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm[5:4]*16
```

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

//RV64 must have a 16-byte aligned sp
if (misa.MXL==2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32 might only need 8-byte aligned SP (EABI)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) bytes=4; else bytes=8;

addr=sp-bytes;
switch(bytes) {
  4:  asm("sw ra, 0(addr)");
  8:  asm("sd ra, 0(addr)");
}
for(i=31;i>=0;i--)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    addr-=bytes;
    switch(bytes) {
      4:  asm("sw s[i], 0(addr)");
      8:  asm("sd s[i], 0(addr)");
    }
  }
}
//The sequence must be uninterruptible from this point
if (areg_list[a0]) asm("mv s0, a0");
if (areg_list[a1]) asm("mv s1, a1");
if (areg_list[a2]) asm("mv s2, a2");

sp+=stack_adjustment; //decrement
```

RV32 Assembly examples

```
c.push  {ra, s0-s2}, {a0-a2}, -64
```

Encoding: *rlist2*=3, *spimm*=3

Equivalent sequence:

```
sw  s2, -4(sp);
sw  s1, -8(sp);
sw  s0, -12(sp);
sw  ra, -16(sp);
mv  s0, a0
mv  s1, a1
mv  s2, a2
addi sp, sp, -64;
```

```
c.push {ra, s0-s1}, {a0-a1}, -32
```

Encoding: *rlist2*=2, *spimm*=1

Equivalent sequence:

```
sw  s1, -4(sp);
sw  s0, -8(sp);
sw  ra, -12(sp);
mv  s0, a0
mv  s1, a1
addi sp, sp, -32;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.pusha.l

Synopsis

Push 5 to 13 registers to stack memory and move argument registers into saved registers, 16-bit encoding

Mnemonic

c.push *{reg_list_5_13}, {areg_list}, -stack_adj*

Encoding (RV32, RV64)

15	14	13	12	11	10	9–7	6	5	4–2	1	0
1	0	1	0	1	0	spimm[6:4]	1	0	rlist3	1	0
FUNCT6									reg_list_5_13	OP=C2	

Syntax

```
c.push {<reg_list_5_13> | <xreg_list_5_13>}, -<stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_5_13> ::= <ra> "," <s0-sN> (where N is 3,4,5,6,7,8,9,11)

if (<reg_list_5_13>=="ra, s0-sN")  <xreg_list_5_13>="x1, x8-x9, x18-xM"
  (where M=N+16 and N is 3,4,5,6,7,8,9,11)

RV32:
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[32|48|..|528] (where N is 3-6)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[48|64|..|544] (where N is 7-9)
if (<reg_list_5_13>=="ra, s0-s11")  <stack_adj>=[64|80|..|560]

RV64:
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 48| 64|..|544] (where N is 3,4)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 64| 80|..|560] (where N is 5,6)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 80| 96|..|576] (where N is 7,8)
if (<reg_list_5_13>=="ra, s0-s9")   <stack_adj>=[ 96|112|..|576]
if (<reg_list_5_13>=="ra, s0-s11")  <stack_adj>=[112|128|..|592]
```

```
RV32/RV64:
if (<reg_list_5_13>=="ra, s0-sN")  <areg_list>="a0-aP"
  (where if (N<4) P=N; else P=3;)
```

Description

This instruction pushes (stores) the registers in *reg_list_5_13* to stack memory, moves *areg_list* into similarly numbered *s* registers, and then adjusts the stack pointer by -*stack_adj*.

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

For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist3* and *spimm* fields in the encoding are as shown below.

Table 2. rlist3 decoding

rlist3	reg_list_5_13	stack_adj_base	
		RV32	RV64
0	ra, s0-s3	32	48
1	ra, s0-s4	32	48
2	ra, s0-s5	32	64
3	ra, s0-s6	32	64
4	ra, s0-s7	48	80
5	ra, s0-s8	48	80
6	ra, s0-s9	48	96
7	ra, s0-s11	64	112

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_5_13*.

spimm is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm[6:4]*16
```

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists.

Operation

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

//RV64 must have a 16-byte aligned sp
if (misa.MXL==2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32 might only need 8-byte aligned SP (EABI)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) bytes=4; else bytes=8;

addr=sp-bytes;
switch(bytes) {
  4:  asm("sw ra, 0(addr)");
  8:  asm("sd ra, 0(addr)");
}
for(i=31;i>=0;i--)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    addr-=bytes;
    switch(bytes) {
      4:  asm("sw s[i], 0(addr)");
      8:  asm("sd s[i], 0(addr)");
    }
  }
}
//The sequence must be uninterruptible from this point
if (areg_list[a0]) asm("mv s0, a0");
if (areg_list[a1]) asm("mv s1, a1");
if (areg_list[a2]) asm("mv s2, a2");
if (areg_list[a3]) asm("mv s3, a3");

sp+=stack_adjustment; //decrement
```

RV32 Assembly examples

```
c.push  {ra, s0-s3}, {a0-a3}, -64
```

Encoding: *rlist3*=0, *spimm*=2

Equivalent sequence:

```
sw  s3, -4(sp);
sw  s2, -8(sp);
sw  s1, -12(sp);
sw  s0, -16(sp);
sw  ra, -20(sp);
mv  s0, a0
mv  s1, a1
mv  s2, a2
mv  s3, a3
addi sp, sp, -64;
```

```
c.push {ra, s0-s11}, {a0-a3}, -128
```

Encoding: *rlist3*=7, *spimm*=4

Equivalent sequence:

```
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);
mv  s0, a0
mv  s1, a1
mv  s2, a2
mv  s3, a3
addi sp, sp, -128;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.push

Synopsis

Push 1 to 4 registers to stack memory, 16-bit encoding

Mnemonic

c.push *{reg_list_1_4}, -stack_adj*

Encoding (RV32, RV64)

15					10	9	8	7	6		4	3	2	1	0
1	0	1	0	1	0	1	spimm[5:4]		0	1	1	rlist2		1	0
FUNCT6												reg_list_1_4		OP=C2	

Syntax

```
c.push {<reg_list_1_4> | <xreg_list_1_4>}, -<stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_1_4> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2)

if (<reg_list_1_4>=="ra")          <xreg_list_1_4>="x1"
if (<reg_list_1_4>=="ra, s0")      <xreg_list_1_4>="x1, x8"
if (<reg_list_1_4>=="ra, s0-s1")   <xreg_list_1_4>="x1, x8-x9"
if (<reg_list_1_4>=="ra, s0-s2")   <xreg_list_1_4>="x1, x8-x9, x18"

RV32:
<stack_adj>=[16|32|48|64]

RV64:
if (<reg_list_1_4>=="ra")          <stack_adj>=[16|32|48|64|96]
if (<reg_list_1_4>=="ra, s0")      <stack_adj>=[16|32|48|64|96]
if (<reg_list_1_4>=="ra, s0-s1")   <stack_adj>=[32|48|64|96|112]
if (<reg_list_1_4>=="ra, s0-s2")   <stack_adj>=[32|48|64|96|112]
```

Description

This instruction pushes (stores) the registers in *reg_list_1_4* to stack memory, and then adjusts the stack pointer by *-stack_adj*.

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

For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist2* field in the encoding are as shown below.

Table 3. rlist2 decoding

rlist2	reg_list_1_4	stack_adj_base	
		RV32	RV64
0	ra	16	16
1	ra, s0	16	16
2	ra, s0-s1	16	32
3	ra, s0-s2	16	32

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_1_4*.

spimm is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm[5:4]*16
```

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

//RV64 must have a 16-byte aligned sp
if (misa.MXL==2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32 might only need 8-byte aligned SP (EABI)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) bytes=4; else bytes=8;

addr=sp-bytes;
switch(bytes) {
  4:  asm("sw ra, 0(addr)");
  8:  asm("sd ra, 0(addr)");
}
for(i=31;i>=0;i--)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    addr-=bytes;
    switch(bytes) {
      4:  asm("sw s[i], 0(addr)");
      8:  asm("sd s[i], 0(addr)");
    }
  }
}
sp+=stack_adjustment; //decrement
```

RV32 Assembly examples

```
c.push  {ra, s0-s2}, -64
```

Encoding: *rlist2*=3, *spimm*=3

Equivalent sequence:

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

```
c.push {ra, s0-s1}, -32
```

Encoding: *rlist2*=2, *spimm*=1

Equivalent sequence:

```
sw  s1, -4(sp);
sw  s0, -8(sp);
sw  ra, -12(sp);
addi sp, sp, -32;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.push.l

Synopsis

Push 5 to 13 registers to stack memory, 16-bit encoding

Mnemonic

c.push *{reg_list_5_13}, -stack_adj*

Encoding (RV32, RV64)

15					10	9	7	6	5	4	2	1	0
1	0	1	0	1	0	spimm[6:4]		1	1	rlist3		1	0
FUNCT6										reg_list_5_13		OP=C2	

Syntax

```
c.push {<reg_list_5_13> | <xreg_list_5_13>}, -<stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_5_13> ::= <ra> "," <s0-sN> (where N is 3,4,5,6,7,8,9,11)

if (<reg_list_5_13>=="ra, s0-sN")  <xreg_list_5_13>="x1, x8-x9, x18-xM"
  (where M=N+16 and N is 3,4,5,6,7,8,9,11)

RV32:
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[32|48|..|528] (where N is 3-6)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[48|64|..|544] (where N is 7-9)
if (<reg_list_5_13>=="ra, s0-s11")  <stack_adj>=[64|80|..|560]

RV64:
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 48| 64|..|544] (where N is 3,4)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 64| 80|..|560] (where N is 5,6)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 80| 96|..|576] (where N is 7,8)
if (<reg_list_5_13>=="ra, s0-s9")   <stack_adj>=[ 96|112|..|576]
if (<reg_list_5_13>=="ra, s0-s11")  <stack_adj>=[112|128|..|592]
```

Description

This instruction pushes (stores) the registers in *reg_list_5_13* to stack memory, and then adjusts the stack pointer by *-stack_adj*.

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

For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist3* and *spimm* fields in the encoding are as shown below.

Table 4. rlist3 decoding

rlist3	reg_list_5_13	stack_adj_base	
		RV32	RV64
0	ra, s0-s3	32	48
1	ra, s0-s4	32	48
2	ra, s0-s5	32	64
3	ra, s0-s6	32	64
4	ra, s0-s7	48	80
5	ra, s0-s8	48	80
6	ra, s0-s9	48	96
7	ra, s0-s11	64	112

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_5_13*.

spimm is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm[6:4]*16
```

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists.

Operation

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

//RV64 must have a 16-byte aligned sp
if (misa.MXL==2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32 might only need 8-byte aligned SP (EABI)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) bytes=4; else bytes=8;

addr=sp-bytes;
switch(bytes) {
  4:  asm("sw ra, 0(addr)");
  8:  asm("sd ra, 0(addr)");
}
for(i=31;i>=0;i--)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    addr-=bytes;
    switch(bytes) {
      4:  asm("sw s[i], 0(addr)");
      8:  asm("sd s[i], 0(addr)");
    }
  }
}
sp+=stack_adjustment; //decrement
```

RV32 Assembly examples

```
c.push  {ra, s0-s3}, -64
```

Encoding: *rlist3*=0, *spimm*=2

Equivalent sequence:

```
sw  s3, -4(sp);
sw  s2, -8(sp);
sw  s1, -12(sp);
sw  s0, -16(sp);
sw  ra, -20(sp);
addi sp, sp, -64;
```

```
c.push {ra, s0-s11}, -128
```

Encoding: *rlist3*=7, *spimm*=4

Equivalent sequence:

```
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, -128;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.pop

Synopsis

Pop 1 to 4 registers from stack memory, 16-bit encoding

Mnemonic

c.pop *{reg_list_1_4}, stack_adj*

Encoding (RV32, RV64)

15					10	9	8	7	6		4	3	2	1	0
1	0	1	0	1	1	0	spimm[5:4]		0	0	1	rlist2		1	0
FUNCT6												reg_list_1_4		OP=C2	

Syntax

```
c.pop {<reg_list_1_4> | <xreg_list_1_4>}, <stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_1_4> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2)

if (<reg_list_1_4>=="ra")          <xreg_list_1_4>="x1"
if (<reg_list_1_4>=="ra, s0")      <xreg_list_1_4>="x1, x8"
if (<reg_list_1_4>=="ra, s0-s1")   <xreg_list_1_4>="x1, x8-x9"
if (<reg_list_1_4>=="ra, s0-s2")   <xreg_list_1_4>="x1, x8-x9, x18"

RV32:
<stack_adj>=[16|32|48|64]

RV64:
if (<reg_list_1_4>=="ra")          <stack_adj>=[16|32|48|64|96]
if (<reg_list_1_4>=="ra, s0")      <stack_adj>=[16|32|48|64|96]
if (<reg_list_1_4>=="ra, s0-s1")   <stack_adj>=[32|48|64|96|112]
if (<reg_list_1_4>=="ra, s0-s2")   <stack_adj>=[32|48|64|96|112]
```

Description

This instruction pop (loads) the registers in *reg_list_1_4* from stack memory, and then adjusts the stack pointer by *stack_adj*.

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

For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist2* field in the encoding are as shown below.

Table 5. rlist2 decoding

rlist2	**reg_list_1_4**	**stack_adj_base**	
		RV32	RV64
0	ra	16	16
1	ra, s0	16	16
2	ra, s0-s1	16	32
3	ra, s0-s2	16	32

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_1_4*.

spimm is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm[5:4]*16
```

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

//RV64 must have a 16-byte aligned sp
if (misa.MXL==2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32 might only need 8-byte aligned SP (EABI)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) bytes=4; else bytes=8;

addr=sp+stack_adjustment-bytes;
switch(bytes) {
  4:  asm("lw ra, 0(addr)");
  8:  asm("ld ra, 0(addr)");
}
for(i=31;i>=0;i--)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    addr-=bytes;
    switch(bytes) {
      4:  asm("lw s[i], 0(addr)");
      8:  asm("ld s[i], 0(addr)");
    }
  }
}
//The sequence must be uninterruptible from this point
sp+=stack_adjustment; //increment
```

RV32 Assembly examples

```
c.pop   {ra, s0-s2}, 48
```

Encoding: *rlist2*=3, *spimm*=2

Equivalent sequence:

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

```
c.pop   {ra}, 16
```

Encoding: *rlist2*=0, *spimm*=0

Equivalent sequence:

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

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.popret

Synopsis

Pop 1 to 4 registers from stack memory and return, 16-bit encoding

Mnemonic

c.popret *{reg_list_1_4}, stack_adj*

Encoding (RV32, RV64)

15					10	9	8	7	6		4	3	2	1	0
1	0	1	0	1	1	0	spimm[5:4]		0	1	1	rlist2		1	0
FUNCT6												reg_list_1_4		OP=C2	

Syntax

```
c.popret {<reg_list_1_4> | <xreg_list_1_4>}, <stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_1_4> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2)

if (<reg_list_1_4>=="ra")          <xreg_list_1_4>="x1"
if (<reg_list_1_4>=="ra, s0")      <xreg_list_1_4>="x1, x8"
if (<reg_list_1_4>=="ra, s0-s1")   <xreg_list_1_4>="x1, x8-x9"
if (<reg_list_1_4>=="ra, s0-s2")   <xreg_list_1_4>="x1, x8-x9, x18"

RV32:
<stack_adj>=[16|32|48|64]

RV64:
if (<reg_list_1_4>=="ra")          <stack_adj>=[16|32|48|64|96]
if (<reg_list_1_4>=="ra, s0")      <stack_adj>=[16|32|48|64|96]
if (<reg_list_1_4>=="ra, s0-s1")   <stack_adj>=[32|48|64|96|112]
if (<reg_list_1_4>=="ra, s0-s2")   <stack_adj>=[32|48|64|96|112]
```

Description

This instruction pop (loads) the registers in *reg_list_1_4* from stack memory, adjusts the stack pointer by *stack_adj* and then returns.

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

For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist2* field in the encoding are as shown below.

Table 6. rlist2 decoding

rlist2	**reg_list_1_4**	**stack_adj_base**	
		RV32	RV64
0	ra	16	16
1	ra, s0	16	16
2	ra, s0-s1	16	32
3	ra, s0-s2	16	32

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_1_4*.

spimm is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm[5:4]*16
```

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

//RV64 must have a 16-byte aligned sp
if (misa.MXL==2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32 might only need 8-byte aligned SP (EABI)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) bytes=4; else bytes=8;

addr=sp+stack_adjustment-bytes;
switch(bytes) {
  4:  asm("lw ra, 0(addr)");
  8:  asm("ld ra, 0(addr)");
}
for(i=31;i>=0;i--)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    addr-=bytes;
    switch(bytes) {
      4:  asm("lw s[i], 0(addr)");
      8:  asm("ld s[i], 0(addr)");
    }
  }
}
//The sequence must be uninterruptible from this point
sp+=stack_adjustment; //increment
asm("ret");
```

RV32 Assembly examples

```
c.popret {ra, s0-s2}, 48
```

Encoding: *rlist2*=3, *spimm*=2

Equivalent sequence:

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

```
c.popret {ra}, 16
```

Encoding: *rlist2*=0, *spimm*=0

Equivalent sequence:

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

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.popret.l

Synopsis

Pop 5 to 13 registers from stack memory and return, 16-bit encoding

Mnemonic

c.popret *{reg_list_5_13}*, *stack_adj*

Encoding (RV32, RV64)

15					10	9 … 7	6	5	4 … 2	1	0
1	0	1	0	1	1	spimm[6:4]	1	0	rlist3	1	0
FUNCT6									reg_list_5_13	OP=C2	

Syntax

```
c.popret {<reg_list_5_13> | <xreg_list_5_13>}, -<stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_5_13> ::= <ra> "," <s0-sN> (where N is 3,4,5,6,7,8,9,11)

if (<reg_list_5_13>=="ra, s0-sN")  <xreg_list_5_13>="x1, x8-x9, x18-xM"
  (where M=N+16 and N is 3,4,5,6,7,8,9,11)

RV32:
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[32|48|..|528] (where N is 3-6)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[48|64|..|544] (where N is 7-9)
if (<reg_list_5_13>=="ra, s0-s11")  <stack_adj>=[64|80|..|560]

RV64:
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 48| 64|..|544] (where N is 3,4)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 64| 80|..|560] (where N is 5,6)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 80| 96|..|576] (where N is 7,8)
if (<reg_list_5_13>=="ra, s0-s9")   <stack_adj>=[ 96|112|..|576]
if (<reg_list_5_13>=="ra, s0-s11")  <stack_adj>=[112|128|..|592]
```

Description

This instruction pop (loads) the registers in *reg_list_5_13* from stack memory, adjusts the stack pointer by *stack_adj* and then returns.

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

For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist3* and *spimm* fields in the encoding are as shown below.

Table 7. rlist3 decoding

rlist3	reg_list_5_13	stack_adj_base	
		RV32	RV64
0	ra, s0-s3	32	48
1	ra, s0-s4	32	48
2	ra, s0-s5	32	64
3	ra, s0-s6	32	64
4	ra, s0-s7	48	80
5	ra, s0-s8	48	80
6	ra, s0-s9	48	96
7	ra, s0-s11	64	112

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_5_13*.

spimm is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm[6:4]*16
```

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists.

Operation

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

//RV64 must have a 16-byte aligned sp
if (misa.MXL==2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32 might only need 8-byte aligned SP (EABI)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) bytes=4; else bytes=8;

addr=sp+stack_adjustment-bytes;
switch(bytes) {
  4:  asm("lw ra, 0(addr)");
  8:  asm("ld ra, 0(addr)");
}
for(i=31;i>=0;i--)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    addr-=bytes;
    switch(bytes) {
      4:  asm("lw s[i], 0(addr)");
      8:  asm("ld s[i], 0(addr)");
    }
  }
}
//The sequence must be uninterruptible from this point
sp+=stack_adjustment; //increment
asm("ret");
```

RV32 Assembly examples

```
c.popret {ra, s0-s3}, 48
```

Encoding: *rlist2*=0, *spimm*=1

Equivalent sequence:

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

```
c.popret {ra, s0-s4}, 64
```

Encoding: *rlist2*=1, *spimm*=2

Equivalent sequence:

```
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;
ret;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.popretz

Synopsis

Pop 1 to 4 registers from stack memory and return zero, 16-bit encoding

Mnemonic

c.popretz *{reg_list_1_4}, stack_adj*

Encoding (RV32, RV64)

15	14	13	12	11	10	9	8–7	6	5	4	3–2	1	0
1	0	1	0	1	1	1	spimm[5:4]	0	1	1	rlist2	1	0
FUNCT6											reg_list_1_4	OP=C2	

Syntax

```
c.popretz {<reg_list_1_4> | <xreg_list_1_4>}, <stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_1_4> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2)

if (<reg_list_1_4>=="ra")           <xreg_list_1_4>="x1"
if (<reg_list_1_4>=="ra, s0")       <xreg_list_1_4>="x1, x8"
if (<reg_list_1_4>=="ra, s0-s1")    <xreg_list_1_4>="x1, x8-x9"
if (<reg_list_1_4>=="ra, s0-s2")    <xreg_list_1_4>="x1, x8-x9, x18"

RV32:
<stack_adj>=[16|32|48|64]

RV64:
if (<reg_list_1_4>=="ra")           <stack_adj>=[16|32|48|64|96]
if (<reg_list_1_4>=="ra, s0")       <stack_adj>=[16|32|48|64|96]
if (<reg_list_1_4>=="ra, s0-s1")    <stack_adj>=[32|48|64|96|112]
if (<reg_list_1_4>=="ra, s0-s2")    <stack_adj>=[32|48|64|96|112]
```

Description

This instruction pop (loads) the registers in *reg_list_1_4* from stack memory, adjusts the stack pointer by *stack_adj*, moves zero into a0 and then returns.

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

For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist2* field in the encoding are as shown below.

Table 8. rlist2 decoding

rlist2	reg_list_1_4	stack_adj_base	
		RV32	RV64
0	ra	16	16
1	ra, s0	16	16
2	ra, s0-s1	16	32
3	ra, s0-s2	16	32

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_1_4*.

spimm is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm[5:4]*16
```

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

//RV64 must have a 16-byte aligned sp
if (misa.MXL==2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32 might only need 8-byte aligned SP (EABI)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) bytes=4; else bytes=8;

addr=sp+stack_adjustment-bytes;
switch(bytes) {
  4:  asm("lw ra, 0(addr)");
  8:  asm("ld ra, 0(addr)");
}
for(i=31;i>=0;i--)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    addr-=bytes;
    switch(bytes) {
      4:  asm("lw s[i], 0(addr)");
      8:  asm("ld s[i], 0(addr)");
    }
  }
}
//The sequence must be uninterruptible from this point
asm("li a0, 0");
sp+=stack_adjustment; //increment
asm("ret";
```

RV32 Assembly examples

```
c.popretz {ra, s0-s2}, 48
```

Encoding: *rlist2*=3, *spimm*=2 Equivalent sequence:

```
lw   s2, 44(sp);
lw   s1, 40(sp);
lw   s0, 36(sp);
lw   ra, 32(sp);
addi sp, sp, 48;
li a0, 0;
ret;
```

```
c.popretz {ra}, 16
```

Encoding: *rlist2*=0, *spimm*=0

Equivalent sequence:

```
lw   ra, 12(sp);
addi sp, sp, 16;
li a0, 0;
ret;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.popretz.l

Synopsis

Pop 5 to 13 registers from stack memory and return zero, 16-bit encoding

Mnemonic

c.popretz *{reg_list_5_13}, stack_adj*

Encoding (RV32, RV64)

15					10	9 7	6	5	4 2	1	0
1	0	1	0	1	1	spimm[6:4]	1	1	rlist3	1	0
FUNCT6									reg_list_5_13	OP=C2	

Syntax

```
c.popretz {<reg_list_5_13> | <xreg_list_5_13>}, -<stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_5_13> ::= <ra> "," <s0-sN> (where N is 3,4,5,6,7,8,9,11)

if (<reg_list_5_13>=="ra, s0-sN")  <xreg_list_5_13>="x1, x8-x9, x18-xM"
  (where M=N+16 and N is 3,4,5,6,7,8,9,11)

RV32:
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[32|48|..|528] (where N is 3-6)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[48|64|..|544] (where N is 7-9)
if (<reg_list_5_13>=="ra, s0-s11")  <stack_adj>=[64|80|..|560]

RV64:
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 48| 64|..|544] (where N is 3,4)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 64| 80|..|560] (where N is 5,6)
if (<reg_list_5_13>=="ra, s0-sN")   <stack_adj>=[ 80| 96|..|576] (where N is 7,8)
if (<reg_list_5_13>=="ra, s0-s9")   <stack_adj>=[ 96|112|..|576]
if (<reg_list_5_13>=="ra, s0-s11")  <stack_adj>=[112|128|..|592]
```

Description

This instruction pop (loads) the registers in *reg_list_5_13* from stack memory, adjusts the stack pointer by *stack_adj*, moves zero into a0 and then returns.

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

For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist3* and *spimm* fields in the encoding are as shown below.

Table 9. rlist3 decoding

rlist3	reg_list_5_13	stack_adj_base	
		RV32	RV64
0	ra, s0-s3	32	48
1	ra, s0-s4	32	48
2	ra, s0-s5	32	64
3	ra, s0-s6	32	64
4	ra, s0-s7	48	80
5	ra, s0-s8	48	80
6	ra, s0-s9	48	96
7	ra, s0-s11	64	112

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_5_13*.

spimm is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm[6:4]*16
```

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists.

Operation

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

//RV64 must have a 16-byte aligned sp
if (misa.MXL==2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32 might only need 8-byte aligned SP (EABI)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) bytes=4; else bytes=8;

addr=sp+stack_adjustment-bytes;
switch(bytes) {
  4:  asm("lw ra, 0(addr)");
  8:  asm("ld ra, 0(addr)");
}
for(i=31;i>=0;i--)  {
  //if register i is in xreg_list
  if (xreg_list[i]) {
    addr-=bytes;
    switch(bytes) {
      4:  asm("lw s[i], 0(addr)");
      8:  asm("ld s[i], 0(addr)");
    }
  }
}
//The sequence must be uninterruptible from this point
asm("li a0, 0");
sp+=stack_adjustment; //increment
asm("ret";
```

RV32 Assembly examples

```
c.popretz {ra, s0-s3}, 48
```

Encoding: *rlist2*=0, *spimm*=1

Equivalent sequence:

```
lw   s3, 44(sp);
lw   s2, 40(sp);
lw   s1, 36(sp);
lw   s0, 32(sp);
lw   ra, 32(sp);
addi sp, sp, 48;
li a0, 0;
ret;
```

```
c.popretz {ra, s0-s4}, 64
```

Encoding: *rlist2*=1, *spimm*=2

Equivalent sequence:

```
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;
li a0, 0;
ret;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.jt

Synopsis

jump via table without link, 16-bit encoding

Mnemonic

c.jt *#index*

Encoding (RV32, RV64)

15					10	9 … 2	1	0
1	0	1	0	0	0	index8	1	0
FUNCT6						index8<64	OP=C2	

NOTE For this encoding to decode as *c.jt*, *index8<64*.

Syntax

```
c.jt #index
```

Description

This instruction is used to dereference a table of PCs, and then jumps without linking to the dereferenced PC.

For further information see Table Jump Instructions.

Prerequisites

The C-extension must also be configured.

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
# Mem is byte indexed
# "index8" is the field from the encoding, not "index" passed to the C.TBLJ* in
the assembler
# which is formed below

if (OPCODE=="C.JALT") {
  index = index8 - 64;
} else {
  index = index8;
}
switch(XLEN) {
  32:  table_address[XLEN-1:0] = JVT.base + index<<2;
  64:  table_address[XLEN-1:0] = JVT.base + index<<3;
}

//check for debug mode entry, trigger with timing=0 and action=1, haltreq or step
if ((debug_trigger(table_address) && MCONTROL.timing==0 && MCONTROL.action==1) ||
    external_debug_haltreq() || DCSR.step==1) {
  DPC        = current_PC;
  DCSR.cause = DCSR.step==1 ? 4 : external_debug_haltreq() ? 3 : 2;
  enter_debug_mode();
//check for breakpoint trigger which takes an exception with timing=0
} else if ((debug_trigger(table_address) && MCONTROL.timing==0) ||
            !can_access_instruction_memory(table_address)) {
  MEPC   = current_PC;
  MTVAL  = table_address;
  MCAUSE = debug_trigger(table_address) ? BREAKPOINT : INSTRUCTION_ACCESS_FAULT;
  take_exception();
} else {
  //access the jump table
  switch(XLEN) {
    32:  LW target_address, InstMemory[table_address][XLEN-1:0];
    64:  LD target_address, InstMemory[table_address][XLEN-1:0];
  }

  //don't use haltreq or step here, only check the addresses
  //check for table_address after reading if timing=1
  if (debug_trigger(table_address) && MCONTROL.timing==1 && MCONTROL.action==1) {
    DPC        = current_PC;
    DCSR.cause = 2;
    enter_debug_mode();
  } else if (debug_trigger(table_address) && MCONTROL.timing==1) {
```

```
    MEPC        = current_PC;
    MTVAL       = table_address;
    MCAUSE      = BREAKPOINT;
    take_exception();
  } else if ((debug_trigger(target_address) && MCONTROL.timing==0 &&
MCONTROL.action==1) {
    DPC         = target_address;
    DCSR.cause = 2;
    enter_debug_mode();
  } else if (((debug_trigger(target_address) && MCONTROL.timing==0) ||
               !can_access_instruction_memory(target_address)) {
    MEPC        = target_address;
    MTVAL       = target_address;
    MCAUSE      = debug_trigger(target_address) ? BREAKPOINT :
INSTRUCTION_ACCESS_FAULT;
    take_exception();
  } else {
    //jump to the target address
    if (OPCODE=="C.JALT") {
      JALR ra, target_address[XLEN-1:0]&~0x1;
    } else {
      JR target_address[XLEN-1:0]&~0x1;
    }
  }
}
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

c.jalt

Synopsis

jump via table and link to ra, 16-bit encoding

Mnemonic

c.jalt *#index*

Encoding (RV32, RV64)

15					10	9	2	1	0
1	0	1	0	0	0	index8		1	0
FUNCT6						index8>=64		OP=C2	

NOTE For this encoding to decode as *c.jalt*, *index8>=64*.

Syntax

```
c.jalt #index
```

NOTE *index* in the assembly syntax is valid from 0-192. *index8* in the encoding is valid from 64-255, so *index* = *index8-64*.

Description

This instruction is used to dereference a table of PCs, and then jumps to the dereferenced PC and links to ra.

For further information see Table Jump Instructions.

Prerequisites

The C-extension must also be configured.

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
# Mem is byte indexed
# "index8" is the field from the encoding, not "index" passed to the C.TBLJ* in
the assembler
# which is formed below

if (OPCODE=="C.JALT") {
  index = index8 - 64;
} else {
  index = index8;
}
switch(XLEN) {
  32:  table_address[XLEN-1:0] = JVT.base + index<<2;
  64:  table_address[XLEN-1:0] = JVT.base + index<<3;
}

//check for debug mode entry, trigger with timing=0 and action=1, haltreq or step
if ((debug_trigger(table_address) && MCONTROL.timing==0 && MCONTROL.action==1) ||
    external_debug_haltreq() || DCSR.step==1) {
  DPC        = current_PC;
  DCSR.cause = DCSR.step==1 ? 4 : external_debug_haltreq() ? 3 : 2;
  enter_debug_mode();
//check for breakpoint trigger which takes an exception with timing=0
} else if ((debug_trigger(table_address) && MCONTROL.timing==0) ||
            !can_access_instruction_memory(table_address)) {
  MEPC   = current_PC;
  MTVAL  = table_address;
  MCAUSE = debug_trigger(table_address) ? BREAKPOINT : INSTRUCTION_ACCESS_FAULT;
  take_exception();
} else {
  //access the jump table
  switch(XLEN) {
    32:  LW target_address, InstMemory[table_address][XLEN-1:0];
    64:  LD target_address, InstMemory[table_address][XLEN-1:0];
  }

  //don't use haltreq or step here, only check the addresses
  //check for table_address after reading if timing=1
  if (debug_trigger(table_address) && MCONTROL.timing==1 && MCONTROL.action==1) {
    DPC        = current_PC;
    DCSR.cause = 2;
    enter_debug_mode();
  } else if (debug_trigger(table_address) && MCONTROL.timing==1) {
```

```
    MEPC        = current_PC;
    MTVAL       = table_address;
    MCAUSE      = BREAKPOINT;
    take_exception();
  } else if ((debug_trigger(target_address) && MCONTROL.timing==0 &&
MCONTROL.action==1) {
    DPC         = target_address;
    DCSR.cause = 2;
    enter_debug_mode();
  } else if (((debug_trigger(target_address) && MCONTROL.timing==0) ||
               !can_access_instruction_memory(target_address)) {
    MEPC        = target_address;
    MTVAL       = target_address;
    MCAUSE      = debug_trigger(target_address) ? BREAKPOINT :
INSTRUCTION_ACCESS_FAULT;
    take_exception();
  } else {
    //jump to the target address
    if (OPCODE=="C.JALT") {
      JALR ra, target_address[XLEN-1:0]&~0x1;
    } else {
      JR target_address[XLEN-1:0]&~0x1;
    }
  }
}
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

JVT CSR

Synopsis

Table jump base vector and control register

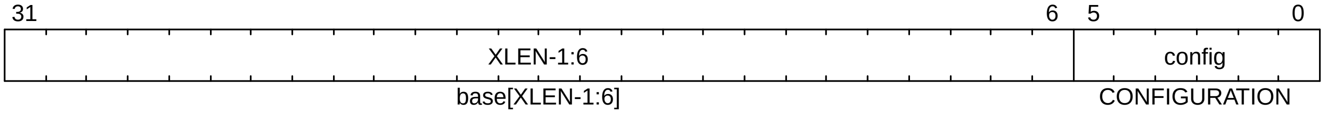
Address

TBD

Permissions

URW

Format (RV32, RV64)



Description

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

Using *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 10. JVT.config definition

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

JVT.config 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
Zces (Zces 0.53.1)	0.53.1	Stable

c.mva01s

Synopsis

Move two s0-s7 registers into a0-a1, 16-bit encoding

Mnemonic

c.mva01s *sreg1, sreg2*

Encoding (RV32, RV64)

15					10	9		7	6	5	4		2	1	0
0	1	0	1	0	0	1	sreg1		1	1		sreg2		1	0
FUNCT6														OP=C2	

Syntax

```
c.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.

Field decoding

Table 11. sreg decoding

sreg*	xreg
0	x8
1	x9
2	x18
3	x19
4	x20
5	x21
6	x22
7	x23

The encoding has two *sreg* number specifiers to save encoding space.

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

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists.

Operation

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

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[sreg1]
X[11] = X[sreg2]
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.53.1	Stable

PUSH/POP register instructions

These instructions are collectively referred to as PUSH/POP:

- c.pusha: push 2 to 4 registers to stack memory and move argument registers into saved registers, 16-bit encoding
- c.pusha.l: push 5 to 13 registers to stack memory and move argument registers into saved registers, 16-bit encoding
- c.push: push 1 to 4 registers to stack memory, 16-bit encoding
- c.push.l: push 5 to 13 registers to stack memory, 16-bit encoding
- c.pop: pop 1 to 4 registers from stack memory, 16-bit encoding
- c.popret: pop 1 to 4 registers from stack memory and return, 16-bit encoding
- c.popret.l: pop 5 to 13 registers from stack memory and return, 16-bit encoding
- c.popretz: pop 1 to 4 registers from stack memory and return zero, 16-bit encoding
- c.popretz.l: pop 5 to 13 registers from stack memory and returns zero, 16-bit encoding

The term PUSH refers to *c.push*, *c.pusha*, *c.push.l* and *c.pusha.l*. The assembly syntax for all of these uses the mnemonic *c.push*.

The term POP refers to *c.pop*.

The term POPRET refers to *c.popret*, *c.popretz*, *c.popret.l* and *c.popretz.l*. The assembly syntax for all of these uses the mnemonics *c.popret/c.popretz*.

Common details for these instructions are in this section.

PUSH/POP overview

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

1. The PUSH instructions
   - push(store) the registers specified in the register list to the stack
   - *c.pusha*, *c.pusha.l* also move the registers in *areg_list* into *s* registers
     - *areg_list* is determined automatically from the register list, it cannot be arbitrarily specified to save encoding space.
   - adjust the stack pointer
2. The POP instruction
   - pops(loads) the registers in the register list from the stack.
   - adjusts the stack pointer
3. The POPRET instructions
   - pop(load) the registers in the register list from the stack.
   - *c.popretz*, *c.popretz.l* also move zero into into *a0* as the return value
   - adjust the stack pointer
   - execute a *ret* instruction.

Example usage

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

```
int function(void *buf, size_t len)
{
        return function2(buf, len);
}
```

compiles with GCC10 to:

```
20405458 <function>:
20405458: 1141                  addi sp,sp,-16     ;#c.pusha(1)
2040545a: c04a                  sw   s2,0(sp)      ;#c.pusha(2)
...
20405464: c422                  sw   s0,8(sp)      ;#c.pusha(3)
20405466: c226                  sw   s1,4(sp)      ;#c.pusha(4)
20405468: c606                  sw   ra,12(sp)     ;#c.pusha(5)
2040546a: 842a                  mv   s0,a0         ;#c.pusha(6)
2040546c: 84ae                  mv   s1,a1         ;#c.pusha(7)
<function body>
20405494: 4501                  li   a0,0          ;#c.popretz(1)
20405496: 40b2                  lw   ra,12(sp)     ;#c.popretz(2)
20405498: 4422                  lw   s0,8(sp)      ;#c.popretz(3)
2040549a: 4492                  lw   s1,4(sp)      ;#c.popretz(4)
2040549c: 4902                  lw   s2,0(sp)      ;#c.popretz(5)
2040549e: 0141                  addi sp,sp,16      ;#c.popretz(6)
204054a0: 8082                  ret                ;#c.popretz(7)
```

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

```
204089ac <function>:
204089ac: f97f72ef              jal  t0,20400942 <__riscv_save_0> ;#c.pusha(1)
...
204089b8: 842a                  mv   s0,a0                        ;#c.pusha(2)
204089ba: 84ae                  mv   s1,a1                        ;#c.pusha(3)
<function_body>
204089e2: 4501                  li   a0,0                         ;#c.popretz(1)
204089e4: f83f706f              j    20400966 <__riscv_restore_0> ;#c.popretz(2)
```

with PUSH/POPRET this reduces to

```
20405458 <function>:
20405458: <16-bit>            c.push    {ra,s0-s2},{a0-a2},-16
<function body>
20405496: <16-bit>            c.popretz {ra,s0-s2}, 16
```

The prologue / epilogue reduce from 28-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 so 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 *c.pusha* has an additional register move included *mv s2, a2* which wasn't in the original prologue. This is included to simplify the encoding and definition of *c.pusha* and will cost some performance.

NOTE POP is used for tail-calling which is not included in this example.

Compiler implementation

The technique used in the initial implementation in LLVM is to let the compiler generate the function prologue and epilogue, and then replace the instruction sequences with the relevant PUSH/POP instructions.

spimm handling

The instrucitons have a restricted range of *spimm* available. If this is insufficient then a seperate *c.addi16sp* can be used to increase the range.

register list handling

c.pop has a restricted register list. If more registers are required then *c.pop* can be used for some of them, and the rest can be explicitly loaded.

The other 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.

areg_list handling

c.pusha, c.pusha.l include *areg_list*. This may not match what was generated by the compiler.

Example: *c.pusha* fits perfectly

In this real world example generated by GCC10, *c.pusha* fits perfectly.

```
00e010b8 <function>:
  e010b8:       1141                    addi    sp,sp,-16 ; #c.pusha
  e010ba:       c422                    sw      s0,8(sp)  ; #c.pusha
  e010bc:       c226                    sw      s1,4(sp)  ; #c.pusha
  e010be:       c04a                    sw      s2,0(sp)  ; #c.pusha
  e010c0:       c606                    sw      ra,12(sp) ; #c.pusha
  e010c2:       842a                    mv      s0,a0     ; #c.pusha
  e010c4:       84ae                    mv      s1,a1     ; #c.pusha
  e010c6:       4908                    lw      a0,16(a0)
  e010c8:       4d8c                    lw      a1,24(a1)
  e010ca:       8932                    mv      s2,a2     ; #c.pusha
  e010cc:       726040ef                jal     ra,e057f2 <function2>
```

this is replaced by

```
00e010b8 <function1>:
  e010b8:       xxxx                    c.push {ra,s0-s2}, {a0-a2}, -16
  e010c6:       4908                    lw      a0,16(a0)
  e010c8:       4d8c                    lw      a1,24(a1)
  e010cc:       726040ef                jal     ra,e057f2 <function2>
```

Example: *areg_ list* doesn't fit

In this other real world example *areg_ list* doesn't fit:

```
00e01126 <function3>:
  e01126:       1101                    addi    sp,sp,-32 ; #c.push.l
  e01128:       ce06                    sw      ra,28(sp) ; #c.push.l
  e0112a:       cc22                    sw      s0,24(sp) ; #c.push.l
  e0112c:       ca26                    sw      s1,20(sp) ; #c.push.l
  e0112e:       c84a                    sw      s2,16(sp) ; #c.push.l
  e01130:       c64e                    sw      s3,12(sp) ; #c.push.l
  e01132:       c452                    sw      s4,8(sp)  ; #c.push.l
  e01134:       c256                    sw      s5,4(sp)  ; #c.push.l
  e01136:       c05a                    sw      s6,0(sp)  ; #c.push.l
  e01138:       0e050363                beqz    a0,e0121e <function3+0xf8>
  e0113c:       8a2a                    mv      s4,a0
  e0113e:       852e                    mv      a0,a1
  e01140:       89ae                    mv      s3,a1
```

In this case, the move instructions are not part of the same basic block so *c.push.l* is selected:

```
00e01126 <function4>:
  e01126:       xxxx                    c.push {ra,s0-s6}, -32
  e01138:       0e050363                beqz    a0,e0121e <function4+0xf8>
  e0113c:       8a2a                    mv      s4,a0
  e0113e:       852e                    mv      a0,a1
  e01140:       89ae                    mv      s3,a1
```

Example: *areg_ list* needs register allocation changes

The next case is where none of the register moves match the *areg_ list* moves because the register allocator in the compiler did not allocate suitable registers:

```
00e01842 <function5>:

  e01e7e:       1101                    addi    sp,sp,-32
  e01e80:       cc22                    sw      s0,24(sp)
  e01e82:       c84a                    sw      s2,16(sp)
  e01e84:       c64e                    sw      s3,12(sp)
  e01e86:       c452                    sw      s4,8(sp)
  e01e88:       c256                    sw      s5,4(sp)
  e01e8a:       ce06                    sw      ra,28(sp)
  e01e8c:       ca26                    sw      s1,20(sp)
  e01e8e:       892a                    mv      s2,a0
  e01e90:       89ae                    mv      s3,a1
  e01e92:       8a32                    mv      s4,a2
  e01e94:       8ab6                    mv      s5,a3
  e01e96:       3f41                    jal     e01e26 <function6>
```

With *c.pusha.l* this becomes:

```
e01e7e <function5>:
  # c.push includes moving {a0-a3} into {s0-s3}
  e01e7e:       1101                    c.push {ra,s0-s5}, {a0-a3}, -32
  e01e8e:       892a                    mv      s2,a0;# <-- switch dest to s0
  e01e90:       89ae                    mv      s3,a1;# <-- switch dest to s1
  e01e92:       8a32                    mv      s4,a2;# <-- switch dest to s2
  e01e94:       8ab6                    mv      s5,a3;# <-- switch dest to s3
  e01e96:       3f41                    jal     e01e26 <function6>
```

In this case all four moves can be deleted if the register allocation can be altered. if the register allocation *cannot* be altered, then *c.push.l* should be used instead.

Example: *areg_ list* partially fits

In this final case, one register move can be deleted and one must be retained unless the register allocation can be changed.

```
00e02368 <function7>:
  e02368:       1141                    addi    sp,sp,-16
  e0236a:       c226                    sw      s1,4(sp)
  e0236c:       03450493                addi    s1,a0,52
  e02370:       c422                    sw      s0,8(sp)
  e02372:       842a                    mv      s0,a0;# <-- delete this one
  e02374:       8526                    mv      a0,s1;# <-- doesn't fit areg_list
  e02376:       c04a                    sw      s2,0(sp)
  e02378:       c606                    sw      ra,12(sp)
  e0237a:       892e                    mv      s2,a1;# <-- switch dest to s1
  e0237c:       df3fd0ef                jal     ra,e0016e <function8>
```

```
00e02368 <function7>:
  e02368:       xxxx                    c.push {ra,s0-s2}, {a0-a2}, -16
  e0236c:       03450493                addi    s1,a0,52
  e02374:       8526                    mv      a0,s1;# <-- doesn't fit areg_list
  e0237a:       892e                    mv      s2,a1;# <-- switch dest to s1
  e0237c:       df3fd0ef                jal     ra,e0016e <function8>
```

In this case one move is deleted, but one remains because unless the target register can be reallocated.

For the smallest code-size the compiler should reallocate the target registers so that the moves in *areg_ list* are not wasted.

PUSH/POP Fault handling

The sequence required to execute the PUSH/POP instruction may be interrupted, or may not be able to start execution for several reasons.

- virtual memory page fault or PMP fault
  - these can be detected before execution, or during execution if the memory addresses cross a page/PMP boundary
  - MTVAL is set to any address which causes the fault
- watchpoint trigger
  - these can be detected before execution, or during execution depending on the trigger type (load data triggers require the sequence to have started executing, for example)
  - MTVAL is set to any address which causes the fault
- external debug halt
  - the halt can treat the whole sequence atomically, or interrupt mid sequence (implementation defined)

- debug halt caused by a trigger
  - same comment as watchpoint trigger above
- load access fault
  - these are detected while the sequence is executing
  - MTVAL is set to the fault address.
- store access fault (precise or imprecise)
  - these may be detected while the sequence is executing, or afterwards if imprecise
  - MTVAL is set to the fault address.
- interrupts
  - these may arrive at any time. An implementation can choose whether to interrupt the sequence or not.

NOTE	MTVAL may be hardwired to zero in an implementation. The section above assumes it is implemented.

In all case MEPC contain the PC of the PUSH/POP instruction, and MCAUSE is set as expected for the type of fault.

For debug halts DPC is set to the PC of the PUSH/POP instruction.

Because some faults can only be detected during the sequence the core implementation is able to recover from the fault and re-execute the sequence. This may involve executing some or all of the loads and stores from the sequence multiple times before the sequence completes (as multiple faults or multiple interrupts are possible).

Therefore correct execution requires that *sp* refers to idempotent memory (also see Non-idempotent memory handling).

Software view of execution

Software view of the PUSH sequence

From a software perspective the PUSH sequence appears as:

- A sequence of stores writing a contiguous block of memory. Any of the bytes may be written multiple times.
- An optional series of register moves
- A stack pointer adjustment

Because the memory is idempotent and the stores are non-overlapping, they may be reordered, grouped into larger accesses, split into smaller access or any combination of these.

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 execution resumes.

The stack pointer adjustment must only be committed once it is certain that all of the stores will complete within triggerring any precise faults (stores may return imprecise bus errors which are received after the instruction has completed execution).

For example:

```
c.push  {ra, s0-s5}, {a0-a3}, -64
```

Appears to software as:

```
# any bytes from sp-1 to sp-28 may be written multiple times before the
instruction completes
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);

# these must only execute once, and will only execute after all stores complete
sucessfully
mv   s0, a0
mv   s1, a1
mv   s2, a2
mv   s3, a3
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, any of which may be executed multiple times
- A stack pointer adjustment
- An optional LI zero into a0
- 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 the interrupt or other fault.

The optional load immediate and stack pointer adjustment must only be committed once it is certain that all of the loads will complete successfully.

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

For example:

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

Appears to software as:

```
# any or all of these load instructions may execute multiple times
lw   s3, 28(sp);
lw   s2, 24(sp);
lw   s1, 20(sp);
lw   s0, 16(sp);
lw   ra, 12(sp);

# must only execute once, will only execute after all loads complete successfully
# all instructions must execute atomically
li a0, 0
addi sp, sp, 32;
ret;
```

Non-idempotent memory handling

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

Error detection

If the core implementation does not have a requirement to support PUSH/POP to non-idempotent memories, and the core can use a PMA to detect that the memory is non-idempotent, then take a load(POP/POPRET) or store (PUSH) access fault exception.

Non-idempotent support

It is possible to support non-idempotent memory. One reason is to re-use PUSH/POP as a restricted form of a load/store multiple instruction to a peripheral, as there is no generic load/store multiple instruction in the RISC-V ISA.

If accessing non-idempotent memory then it is *recommended* to:

1. Not allow interrupts during execution
2. Not allow external debug halt during execution
3. Detect any virtual memory page faults or PMP faults for the whole instruction before starting execution (instead of during the sequence)
4. Not split / merge / reorder the generated memory accesses

It is possible that one of the following will still occur during execution:

1. Watchpoint trigger
2. Load/store access fault

In these cases the core will jump to the debug or exception handler. If execution is required to continue afterwards (so the event is not fatal to the code execution), then the handler is required to do so in software.

By following these rules memory accesses will only ever be issued once, and in the order listed in the SAIL.

It is possible for implementations to follow these restricted rules and to safely access both types of memory. It is also possible for an implementation to use PMAs to detect the memory type and apply different rules, such as only allowing interrupts if accessing cacheable memory, for example.

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.1)	0.52	Stable

Table Jump Instructions

These instructions are collectively referred to as table jump:

- c.jt: jump via table without link, 16-bit encoding
- c.jalt: jump via table and link to ra, 16-bit encoding

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 *c.jt*
- replace 32-bit *jal* ra calls with *c.jalt*
- replace 64-bit *auipc/jalr* calls to fixed locations with *c.jt*
- replace 64-bit *auipc/jalr ra* calls to fixed locations with *c.jalt*
  - The AUIPC+JR/JALR sequence is used because the offset from the PC is out of the ±1MB range.

TBLJALVEC

The base of the table is in the TBLJALVEC CSR (see [csrs-tbljalvec]), each table entry is XLEN bits.

The table entry number is from the *index8* field in the encoding, which controls the link register.

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

Note that the LSB of every jump 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 table

Calls to each function are categorised as shown in Table jump code size saving for each function call replacement.

Table 12. Table jump code size saving for each function call replacement

original sequence	Table Jump saving
J	A*2-(XLEN/8) bytes
AUIPC+JR	B*6-(XLEN/8) bytes
JAL ra	C*2-(XLEN/8) bytes
AUIPC+JALR ra	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_c_jt    = A * 2 + B * 6 - 2*(XLEN-8)
saving_per_function_c_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 table. Only caching a few entries will greatly improve the performance.

Table Jump Fault handling

Table Jump involves two instruction fetches from a single instruction, and either fetch can cause a fault.

The sequence required to execute the table jump instruction may be interrupted, or may not be able to start execution for several reasons.

- virtual memory page fault or PMP fault
  - these can be detected before execution, or during execution if the memory addresses cross a page/PMP boundary
  - MTVAL is set to any address which causes the fault
- watchpoint trigger
  - these can be detected before execution, or during execution depending on the trigger type (load data triggers require the sequence to have started executing, for example)
  - MTVAL is set to any address which causes the fault
- external debug halt
  - the halt can treat the whole sequence atomically, or interrupt mid sequence (implementation defined)
- debug halt caused by a trigger
  - same comment as watchpoint trigger above
- load access fault
  - these are detected while the sequence is executing
  - MTVAL is set to the fault address.
- store access fault (precise or imprecise)
  - these may be detected while the sequence is executing, or afterwards if imprecise
  - MTVAL is set to the fault address.
- interrupts
  - these may arrive at any time. An implementation can choose whether to interrupt the sequence or not.

In all case MEPC contain the PC of the table jump instruction, and MCAUSE is set as expected for the type of fault.

For debug halts DPC is set to the PC of the table jump instruction.

This section gives an overview of the behaviour, the exact operation is documented in the SAIL code for each instruction

- c.jalt SAIL code
- c.jt SAIL code

Included in

Extension	
Minimum version	
Lifecycle state	

Zces (Zces 0.53.1)
0.53.1
Stable