Zces 0.51

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.

All 16-bit encodings are currently reserved for all architectures, and have no conflicts with any existing extensions.

All 32-bit encodings have yet to be allocated.

tblj, tbljal, tbljalm all require tbljalvec CSR, table jump base vector and control register.

RV 32	RV 64	RV 128	Mnemonic	Instruction
✓	✓	✓	c.push {reg_list}, {areg_list}, -sp_adj	c.push: push registers to stack memory, 16-bit encoding
✓	✓	✓	push {reg_list}, {areg_list}, -sp_adj	push: push registers to stack memory, 32-bit encoding
✓	✓	✓	c.pop {reg_list}, sp_adjustment	c.pop: pop registers from the stack, 16-bit encoding
✓	✓	✓	pop {reg_list}, sp_adjustment	pop: pop registers from the stack, 32-bit encoding
✓	✓	✓	c.popret {reg_list}, {ret_val}, sp_adj	c.popret: pop registers and return, 16-bit encoding
✓	✓	✓	popret {reg_list}, {ret_val}, sp_adj	popret: pop registers from the stack and return, 32-bit encoding
✓	✓	✓	tblj #index	c.tblj: table jump without link, 16-bit encoding
✓	✓	✓	tbljal #index	c.tbljal: table jump and link to ra, 16-bit encoding
✓	✓	✓	tbljalm #index	c.tbljalm: table jump and link to t0, 16-bit encoding
✓	✓	✓	c.mva01s07 sreg1, sreg2	c.mva01s07: move two s0-s7 registers into a0-a1, 16-bit encoding

c.push

Synopsis

Push registers to stack memory, 16-bit encoding

Mnemonic

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

Encoding (RV32, RV64, RV128)

15					10	9		7	6	5	4		2	1	0
1	0	0	0	1	1	spimm0_5[6:4]			1	0	rlist3			0	0
FUNCT6											reg_list			OP=C0	

NOTE *spimm0_5* is only valid for *c.push* for values 0-5. Values 6 and 7 map onto *different* encodings.

Syntax

```
c.push {<reg_list_16u> | <xreg_list_16u>}, {<areg_list>), -<stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_16u> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2,3,5,7,11)

if (<reg_list_16u>=="ra")           <xreg_list_16u>="x1"
if (<reg_list_16u>=="ra, s0")       <xreg_list_16u>="x1, x8"
if (<reg_list_16u>=="ra, s0-s1")    <xreg_list_16u>="x1, x8-x9"
if (<reg_list_16u>=="ra, s0-s2")    <xreg_list_16u>="x1, x8-x9, x18"
if (<reg_list_16u>=="ra, s0-sN")    <xreg_list_16u>="x1, x8-x9, x18-xM" (where
M=N+16 and N is 3,5,7,11)

if (<reg_list_16u>=="ra")           <areg_list>=""
if (<reg_list_16u>=="ra, s0")       <areg_list>="a0"
if (<reg_list_16u>=="ra, s0-sN")    <areg_list>="a0-aP" (where if (N<4) P=N; else
P=3;)

if (<reg_list_16u>=="ra")           <stack_adj>=[16|32|48|64|96]
if (<reg_list_16u>=="ra, s0")       <stack_adj>=[16|32|48|64|96]
if (<reg_list_16u>=="ra, s0-s1")    <stack_adj>=[16|32|48|64|96]
if (<reg_list_16u>=="ra, s0-s2")    <stack_adj>=[16|32|48|64|96]
if (<reg_list_16u>=="ra, s0-s3")    <stack_adj>=[32|48|64|96|112]
if (<reg_list_16u>=="ra, s0-s5")    <stack_adj>=[32|48|64|96|112]
if (<reg_list_16u>=="ra, s0-s7")    <stack_adj>=[48|64|96|112|128]
if (<reg_list_16u>=="ra, s0-s11")   <stack_adj>=[64|96|112|128|144]
```

Description

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

PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist3* and *spimm0_5* fields in the encoding are as shown below.

Table 1. rlist3 decoding

rlist3	reg_list_16u	stack_adj_base
0	ra	16
1	ra, s0	16
2	ra, s0-s1	16
3	ra, s0-s2	16
4	ra, s0-s3	32
5	ra, s0-s5	32
6	ra, s0-s7	48
7	ra, s0-s11	64

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_16u*.

spimm_0_5 is used to allocate extra stack space in 16-byte blocks.

The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm0_5[6:4]*16
```

Prerequisites

The C-extension must also be configured.

Operation

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

//RV64/RV128 must have a 16-byte aligned sp
if (misa.MXL>=2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32I might be using the EABI (8-byte alignment) or UABI (16-byte alignment, so
in hardware we can only check for 8)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) {bytes=4;}
if (misa.MXL==2) {bytes=8;}
else             {bytes=16;}
addr=sp-bytes;
switch(bytes) {
  4:  asm("sw ra, 0(addr)");
  8:  asm("sd ra, 0(addr)");
  16: asm("sq 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)");
      16: asm("sq 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
```

Assembly examples

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

Encoding: *rlist3*=5, *spimm0_5[6:4]*=2

Equivalent sequence:

```
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);
mv  s0, a0
mv  s1, a1
mv  s2, a2
mv  s3, a3
addi sp, sp, -64;
```

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

Encoding: *rlist3*=2, *spimm0_5[6:4]*=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 0.51)	0.51	Plan

push

Synopsis

Push registers to stack memory, 32-bit encoding

Mnemonic

push *{reg_list}*, *{<areg_list>}*, *-stack_adj*

Encoding (RV32, RV64, RV128)

31–21	20	19–16	15	14–12	11–7	6–0
TBD	areg	rlist_0_12	TBD	TBD	spimm[8:4]	TBD
FUNCT6		reg_list		funct3		opcode

NOTE *rlist_0_12* is only valid for *push* for values 0-12. Values 13-15 map onto *different* encodings.

Syntax

```
push {<reg_list_32u> | <xreg_list_32u>}, {<areg_list>}, -<stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_32u> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2,...,11)

if (<reg_list_32u>=="ra")          <xreg_list_32u>="x1"
if (<reg_list_32u>=="ra, s0")      <xreg_list_32u>="x1, x8"
if (<reg_list_32u>=="ra, s0-s1")   <xreg_list_32u>="x1, x8-x9"
if (<reg_list_32u>=="ra, s0-s2")   <xreg_list_32u>="x1, x8-x9, x18"
if (<reg_list_32u>=="ra, s0-sN")   <xreg_list_32u>="x1, x8-x9, x18-xM" (where
M=N+16 and N is 3-11)

if (<reg_list_32u>=="ra")           <areg_list>=""
if (<reg_list_32u>=="ra, s0")       <areg_list>="" | "a0"
if (<reg_list_32u>=="ra, s0-sN")    <areg_list>="" | "a0-aP" (where if (N<4) P=N;
else P=3;)

if (<reg_list_32u>=="ra")           <stack_adj>=[16|32|..|512]
if (<reg_list_32u>=="ra, s0")       <stack_adj>=[16|32|..|512]
if (<reg_list_32u>=="ra, s0-sN")    <stack_adj>=[16|32|..|512] (where N is 1,2)
if (<reg_list_32u>=="ra, s0-sN")    <stack_adj>=[32|48|..|528] (where N is 3,4,5,6)
if (<reg_list_32u>=="ra, s0-sN")    <stack_adj>=[48|64|..|544] (where N is
7,8,9,10)
if (<reg_list_32u>=="ra, s0-s11")   <stack_adj>=[64|96|..|560]
```

Description

This instruction pushes (stores) the registers in *reg_list* to stack memory, and then adjusts the stack pointer by *-stack_adj*. For further information see PUSH/POP Register Instructions.

Prerequisites

The C-extension must also be configured.

Field decoding

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

Table 2. rlist decoding

rlist_0_12	reg_list_32u	stack_adj_base
0	ra	16
1	ra, s0	16
2	ra, s0-s1	16
3	ra, s0-s2	16
4	ra, s0-s3	32
5	ra, s0-s4	32
6	ra, s0-s5	32
7	ra, s0-s6	32
8	ra, s0-s7	48
9	ra, s0-s8	48
10	ra, s0-s9	48
11	ra, s0-s10	48
12	ra, s0-s11	64

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_32u*. *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[8:4]*16
```

Table 3. areg_list decoding

rlist_0_12	areg_list	
	areg=0	areg=1
0	""	""
1	""	a0
2	""	a0-a1
3	""	a0-a2
4-12	""	a0-a3

Prerequisites

The C-extension must also be configured.

Operation

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

//RV64/RV128 must have a 16-byte aligned sp
if (misa.MXL>=2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32I might be using the EABI (8-byte alignment) or UABI (16-byte alignment, so
in hardware we can only check for 8)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) {bytes=4;}
if (misa.MXL==2) {bytes=8;}
else             {bytes=16;}
addr=sp-bytes;
switch(bytes) {
  4:  asm("sw ra, 0(addr)");
  8:  asm("sd ra, 0(addr)");
  16: asm("sq 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)");
      16: asm("sq 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
```

Assembly examples

```
push  {ra, s0-s4}, {a0-a3}, -528
```

Encoding: *rlist*=5, *spimm0_5[8:4]*=0x1f, *areg*=1

Equivalent sequence:

```
sw  s4, -4(sp);
sw  s3, -8(sp);
sw  s2, -12(sp);
sw  s1, -16(sp);
sw  s0, -20(sp);
sw  ra, -24(sp);
mv  s0, a0
mv  s1, a1
mv  s2, a2
mv  s3, a3
addi sp, sp, -528;
```

```
push {ra, s0-s3}, {}, -32
```

Encoding: *rlist3*=2, *spimm[8:4]*=1, *areg*=0

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, -32;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.51)	0.51	Plan

c.popret

Synopsis

Pop registers and return, 16-bit encoding

Mnemonic

c.popret *{reg_list}*, *{ret_val}*, *stack_adj*

Encoding (RV32, RV64, RV128)

15	14	13	12	11	10	9–7	6	5	4–2	1	0
1	0	0	0	1	1	spimm0_5[6:4]	1	ret0	rlist3	0	0
FUNCT6									reg_list	OP=C0	

NOTE *spimm0_5* is only valid for *c.pop* for values 0-5. Values 6 and 7 map onto *different* encodings.

Syntax

```
c.popret {<reg_list_16u> | <xreg_list_16u>}, <stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_16u> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2,3,5,7,11)

if (<reg_list_16u>=="ra")          <xreg_list_16u>="x1"
if (<reg_list_16u>=="ra, s0")      <xreg_list_16u>="x1, x8"
if (<reg_list_16u>=="ra, s0-s1")  <xreg_list_16u>="x1, x8-x9"
if (<reg_list_16u>=="ra, s0-s2")  <xreg_list_16u>="x1, x8-x9, x18"
if (<reg_list_16u>=="ra, s0-sN")  <xreg_list_16u>="x1, x8-x9, x18-xM" (where
M=N+16 and N is 3,5,7,11)

if (<reg_list_16u>=="ra")           <stack_adj>=[16|32|48|64|96]
if (<reg_list_16u>=="ra, s0")       <stack_adj>=[16|32|48|64|96]
if (<reg_list_16u>=="ra, s0-s1")   <stack_adj>=[16|32|48|64|96]
if (<reg_list_16u>=="ra, s0-s2")   <stack_adj>=[16|32|48|64|96]
if (<reg_list_16u>=="ra, s0-s3")   <stack_adj>=[32|48|64|96|112]
if (<reg_list_16u>=="ra, s0-s5")   <stack_adj>=[32|48|64|96|112]
if (<reg_list_16u>=="ra, s0-s7")   <stack_adj>=[48|64|96|112|128]
if (<reg_list_16u>=="ra, s0-s11")  <stack_adj>=[64|96|112|128|144]
```

Description

This instruction pop (loads) the registers in *reg_list* from stack memory, and then adjusts the stack pointer by *stack_adj*. For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist3* and *spimm0_5* fields in the encoding are as shown below.

Table 4. rlist3 decoding

rlist3	reg_list_16u	stack_adj_base
0	ra	16
1	ra, s0	16
2	ra, s0-s1	16
3	ra, s0-s2	16
4	ra, s0-s3	32
5	ra, s0-s5	32
6	ra, s0-s7	48
7	ra, s0-s11	64

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_16u*. *spimm_0_5* is used to allocate extra stack space in 16-byte blocks. The total stack adjustment is calculated as shown.

```
stack_adj = stack_adj_base+spimm0_5[6:4]*16
```

Prerequisites

The C-extension must also be configured.

Operation

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

//RV64/RV128 must have a 16-byte aligned sp
if (misa.MXL>=2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32I might be using the EABI (8-byte alignment) or UABI (16-byte alignment, so
in hardware we can only check for 8)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) {bytes=4;}
if (misa.MXL==2) {bytes=8;}
else             {bytes=16;}
addr=sp+stack_adjustment-bytes;
switch(bytes) {
  4:  asm("lw ra, 0(addr)");
  8:  asm("ld ra, 0(addr)");
  16: asm("lq 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)");
      16: asm("lq s[i], 0(addr)");
    }
  }
}
if (ret_val) {
   switch(ret_val) {
      "0":  asm("li a0, 0");
   }
}
//The sequence must be uninterruptible from this point
sp+=stack_adjustment; //increment
asm("ret");
```

Assembly examples

```
c.popret   {ra, s0-s7}, {0}, 160
```

Encoding: *rlist3*=6, *spimm0_5[6:4]*=7, *ret0*=1

Equivalent sequence:

```
lw   s7, 156(sp);
lw   s6, 152(sp);
lw   s5, 148(sp);
lw   s4, 144(sp);
lw   s3, 140(sp);
lw   s2, 136(sp);
lw   s1, 132(sp);
lw   s0, 128(sp);
lw   ra, 124(sp);
li   a0, 0;
addi sp, sp, 160;
ret
```

```
c.popret   {ra, s0-s7}, {}, 160
```

Encoding: *rlist3*=6, *spimm0_5[6:4]*=7, *ret0*=0

Equivalent sequence:

```
lw   s7, 156(sp);
lw   s6, 152(sp);
lw   s5, 148(sp);
lw   s4, 144(sp);
lw   s3, 140(sp);
lw   s2, 136(sp);
lw   s1, 132(sp);
lw   s0, 128(sp);
lw   ra, 124(sp);
addi sp, sp, 160;
ret
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.51)	0.51	Plan

popret

Synopsis

popret registers, 32-bit encoding

Mnemonic

popret *{reg_list}*, *{retval}*, *stack_adj*

Encoding (RV32, RV64, RV128)

31–22	21–20	19–16	15	14–12	11–7	6–0
TBD	ret_val	rlist_0_12	TBD	TBD	spimm[8:4]	TBD
FUNCT6		reg_list		funct3		opcode

NOTE *rlist_0_12* is only valid for *popret* for values 0-12. Values 13-15 map onto *different* encodings.

Syntax

```
popret {<reg_list_32u> | <xreg_list_32u>}, <stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_32u> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2,...,11)

if (<reg_list_32u>=="ra")          <xreg_list_32u>="x1"
if (<reg_list_32u>=="ra, s0")      <xreg_list_32u>="x1, x8"
if (<reg_list_32u>=="ra, s0-s1")   <xreg_list_32u>="x1, x8-x9"
if (<reg_list_32u>=="ra, s0-s2")   <xreg_list_32u>="x1, x8-x9, x18"
if (<reg_list_32u>=="ra, s0-sN")   <xreg_list_32u>="x1, x8-x9, x18-xM" (where
M=N+16 and N is 3-11)

if (<reg_list_32u>=="ra")           <stack_adj>=[16|32|..|512]
if (<reg_list_32u>=="ra, s0")       <stack_adj>=[16|32|..|512]
if (<reg_list_32u>=="ra, s0-sN")    <stack_adj>=[16|32|..|512] (where N is 1,2)
if (<reg_list_32u>=="ra, s0-sN")    <stack_adj>=[32|48|..|528] (where N is 3,4,5,6)
if (<reg_list_32u>=="ra, s0-sN")    <stack_adj>=[48|64|..|544] (where N is
7,8,9,10)
if (<reg_list_32u>=="ra, s0-s11")   <stack_adj>=[64|96|..|560]
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, and then adjusts the stack pointer by *stack_adj*. For further information see PUSH/POP Register Instructions.

Prerequisites

The C-extension must also be configured.

Field decoding

The mapping from the *ret_val*, *rlist* and *spimm* fields in the encoding are as shown below.

Table 5. rlist decoding

ret_val	reg_list_32u
0	no return value
1	a0=0
2	a0=1
3	a0=-1

Table 6. rlist decoding

rlist3	reg_list_32u	stack_adj_base
0	ra	16
1	ra, s0	16
2	ra, s0-s1	16
3	ra, s0-s2	16
4	ra, s0-s3	32
5	ra, s0-s4	32
6	ra, s0-s5	32
7	ra, s0-s6	32
8	ra, s0-s7	48
9	ra, s0-s8	48
10	ra, s0-s9	48
11	ra, s0-s10	48
12	ra, s0-s11	64

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_32u*. *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[8:4]*16
```

Prerequisites

The C-extension must also be configured.

Operation

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

//RV64/RV128 must have a 16-byte aligned sp
if (misa.MXL>=2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32I might be using the EABI (8-byte alignment) or UABI (16-byte alignment, so
in hardware we can only check for 8)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) {bytes=4;}
if (misa.MXL==2) {bytes=8;}
else             {bytes=16;}
addr=sp+stack_adjustment-bytes;
switch(bytes) {
  4:  asm("lw ra, 0(addr)");
  8:  asm("ld ra, 0(addr)");
  16: asm("lq 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)");
      16: asm("lq s[i], 0(addr)");
    }
  }
}
if (ret_val) {
   switch(ret_val) {
      "0":  asm("li a0, 0");
      "1":  asm("li a0, 1");
      "2":  asm("li a0, -1");
   }
}
//The sequence must be uninterruptible from this point
sp+=stack_adjustment; //increment
asm("ret");
```

Assembly examples

```
popret   {ra, s0-s6}, {0}, 160
```

Encoding: *rlist*=7, *spimm[8:4]*=7, *ret0*=1

Equivalent sequence:

```
lw   s6, 156(sp);
lw   s5, 152(sp);
lw   s4, 148(sp);
lw   s3, 144(sp);
lw   s2, 140(sp);
lw   s1, 136(sp);
lw   s0, 132(sp);
lw   ra, 128(sp);
li   a0, 0;
addi sp, sp, 160;
ret
```

```
popret   {ra, s0-s7}, {-1}, 160
```

Encoding: *rlist*=8, *spimm[8:4]*=7, *ret0*=2

Equivalent sequence:

```
lw   s7, 156(sp);
lw   s6, 152(sp);
lw   s5, 148(sp);
lw   s4, 144(sp);
lw   s3, 140(sp);
lw   s2, 136(sp);
lw   s1, 132(sp);
lw   s0, 128(sp);
lw   ra, 124(sp);
li   a0, -1;
addi sp, sp, 160;
ret
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.51)	0.51	Plan

c.pop

Synopsis

Pop registers, 16-bit encoding

Mnemonic

c.pop *{reg_list}*, *stack_adj*

Encoding (RV32, RV64, RV128)

15							8	7	6	5	4		2	1	0
1	0	0	0	1	1	1	1	spimm[4]	1	0	rlist3			0	0
FUNCT6											reg_list			OP=C0	

NOTE *spimm0_5* is only valid for *c.pop* for values 0-5. Values 6 and 7 map onto *different* encodings.

Syntax

```
c.pop {<reg_list_16u> | <xreg_list_16u>}, <stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_16u> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2,3,5,7,11)

if (<reg_list_16u>=="ra")         <xreg_list_16u>="x1"
if (<reg_list_16u>=="ra, s0")     <xreg_list_16u>="x1, x8"
if (<reg_list_16u>=="ra, s0-s1")  <xreg_list_16u>="x1, x8-x9"
if (<reg_list_16u>=="ra, s0-s2")  <xreg_list_16u>="x1, x8-x9, x18"
if (<reg_list_16u>=="ra, s0-sN")  <xreg_list_16u>="x1, x8-x9, x18-xM" (where
M=N+16 and N is 3,5,7,11)

if (<reg_list_16u>=="ra")          <stack_adj>=[16|32]
if (<reg_list_16u>=="ra, s0")      <stack_adj>=[16|32]
if (<reg_list_16u>=="ra, s0-s1")   <stack_adj>=[16|32]
if (<reg_list_16u>=="ra, s0-s2")   <stack_adj>=[16|32]
if (<reg_list_16u>=="ra, s0-s3")   <stack_adj>=[32|48]
if (<reg_list_16u>=="ra, s0-s5")   <stack_adj>=[32|48]
if (<reg_list_16u>=="ra, s0-s7")   <stack_adj>=[48|64]
if (<reg_list_16u>=="ra, s0-s11")  <stack_adj>=[64|96]
```

Description

This instruction pop (loads) the registers in *reg_list* from stack memory, and then adjusts the stack pointer by *stack_adj*. For further information see PUSH/POP Register Instructions.

Field decoding

The mapping from the *rlist3* and *spimm0_5* fields in the encoding are as shown below.

Table 7. rlist3 decoding

rlist3	reg_list_16u	stack_adj_base
0	ra	16
1	ra, s0	16
2	ra, s0-s1	16
3	ra, s0-s2	16
4	ra, s0-s3	32
5	ra, s0-s5	32
6	ra, s0-s7	48
7	ra, s0-s11	64

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_16u*. *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[4]*16
```

Prerequisites

The C-extension must also be configured.

Operation

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

//RV64/RV128 must have a 16-byte aligned sp
if (misa.MXL>=2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32I might be using the EABI (8-byte alignment) or UABI (16-byte alignment, so
in hardware we can only check for 8)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) {bytes=4;}
if (misa.MXL==2) {bytes=8;}
else             {bytes=16;}
addr=sp+stack_adjustment-bytes;
switch(bytes) {
  4:  asm("lw ra, 0(addr)");
  8:  asm("ld ra, 0(addr)");
  16: asm("lq 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)");
      16: asm("lq s[i], 0(addr)");
    }
  }
}
//The sequence must be uninterruptible from this point
sp+=stack_adjustment; //increment
```

Assembly examples

```
c.pop   {ra, s0-s7}, 160
```

Encoding: *rlist3*=6, *spimm[4]*=0 Equivalent sequence:

```
lw   s7, 44(sp);
lw   s6, 40(sp);
lw   s5, 36(sp);
lw   s4, 32(sp);
lw   s3, 28(sp);
lw   s2, 24(sp);
lw   s1, 20(sp);
lw   s0, 16(sp);
lw   ra, 12(sp);
addi sp, sp, 48;
ret
```

```
c.pop   {ra, s0-s7}, 160
```

Encoding: *rlist3*=6, *spimm[4]*=1

Equivalent sequence:

```
lw   s7, 60(sp);
lw   s6, 56(sp);
lw   s5, 52(sp);
lw   s4, 48(sp);
lw   s3, 44(sp);
lw   s2, 40(sp);
lw   s1, 36(sp);
lw   s0, 32(sp);
lw   ra, 28(sp);
addi sp, sp, 64;
ret
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.51)	0.51	Plan

pop

Synopsis

Pop registers, 32-bit encoding

Mnemonic

pop *{reg_list}, stack_adj*

Encoding (RV32, RV64, RV128)

31–22	21–20	19–16	15	14–12	11–7	6–0
TBD	ret_val	rlist_0_12	TBD	TBD	spimm[8:4]	TBD
FUNCT6		reg_list		funct3		opcode

NOTE *rlist_0_12* is only valid for *pop* for values 0-12. Values 13-15 map onto *different* encodings.

Syntax

```
pop {<reg_list_32u> | <xreg_list_32u>}, <stack_adj>
```

The variables used in the syntax are defined below.

```
<reg_list_32u> ::= <ra> ["," <s0>  | <s0-sN> ] (where N is 1,2,...,11)

if (<reg_list_32u>=="ra")          <xreg_list_32u>="x1"
if (<reg_list_32u>=="ra, s0")      <xreg_list_32u>="x1, x8"
if (<reg_list_32u>=="ra, s0-s1")   <xreg_list_32u>="x1, x8-x9"
if (<reg_list_32u>=="ra, s0-s2")   <xreg_list_32u>="x1, x8-x9, x18"
if (<reg_list_32u>=="ra, s0-sN")   <xreg_list_32u>="x1, x8-x9, x18-xM" (where
M=N+16 and N is 3-11)

if (<reg_list_32u>=="ra")           <stack_adj>=[16|32|..|512]
if (<reg_list_32u>=="ra, s0")       <stack_adj>=[16|32|..|512]
if (<reg_list_32u>=="ra, s0-sN")    <stack_adj>=[16|32|..|512] (where N is 1,2)
if (<reg_list_32u>=="ra, s0-sN")    <stack_adj>=[32|48|..|528] (where N is 3,4,5,6)
if (<reg_list_32u>=="ra, s0-sN")    <stack_adj>=[48|64|..|544] (where N is
7,8,9,10)
if (<reg_list_32u>=="ra, s0-s11")  <stack_adj>=[64|96|..|560]
```

Description

This instruction pops (loads) the registers in *reg_list* from stack memory, and then adjusts the stack pointer by *stack_adj*. For further information see PUSH/POP Register Instructions.

Prerequisites

The C-extension must also be configured.

Field decoding

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

Table 8. rlist decoding

rlist3	reg_list_32u	stack_adj_base
0	ra	16
1	ra, s0	16
2	ra, s0-s1	16
3	ra, s0-s2	16
4	ra, s0-s3	32
5	ra, s0-s4	32
6	ra, s0-s5	32
7	ra, s0-s6	32
8	ra, s0-s7	48
9	ra, s0-s8	48
10	ra, s0-s9	48
11	ra, s0-s10	48
12	ra, s0-s11	64

stack_adj_base covers enough 16-byte blocks of memory to cover the registers in *reg_list_32u*. *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[8:4]*16
```

Prerequisites

The C-extension must also be configured.

Operation

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

//RV64/RV128 must have a 16-byte aligned sp
if (misa.MXL>=2 && sp[3:0]) {take_illegal_instruction_exception();}
//RV32I might be using the EABI (8-byte alignment) or UABI (16-byte alignment, so
in hardware we can only check for 8)
if (misa.MXL==1 && sp[2:0]) {take_illegal_instruction_exception();}

if (misa.MXL==1) {bytes=4;}
if (misa.MXL==2) {bytes=8;}
else             {bytes=16;}
addr=sp+stack_adjustment-bytes;
switch(bytes) {
  4:  asm("lw ra, 0(addr)");
  8:  asm("ld ra, 0(addr)");
  16: asm("lq 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)");
      16: asm("lq s[i], 0(addr)");
    }
  }
}
//The sequence must be uninterruptible from this point
sp+=stack_adjustment; //increment
```

Assembly examples

```
pop   {ra, s0-s6}, 160
```

Encoding: *rlist*=7, *spimm[8:4]*=7

Equivalent sequence:

```
lw   s6, 156(sp);
lw   s5, 152(sp);
lw   s4, 148(sp);
lw   s3, 144(sp);
lw   s2, 140(sp);
lw   s1, 136(sp);
lw   s0, 132(sp);
lw   ra, 128(sp);
addi sp, sp, 160;
ret
```

```
pop   {ra, s0-s7}, 160
```

Encoding: *rlist*=8, *spimm[8:4]*=7

Equivalent sequence:

```
lw   s7, 156(sp);
lw   s6, 152(sp);
lw   s5, 148(sp);
lw   s4, 144(sp);
lw   s3, 140(sp);
lw   s2, 136(sp);
lw   s1, 132(sp);
lw   s0, 128(sp);
lw   ra, 124(sp);
addi sp, sp, 160;
ret
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.51)	0.51	Plan

c.tblj

Synopsis

table jump, no link, 16-bit encoding

Mnemonic

c.tblj *#index*

Encoding (RV32, RV64, RV128)

15					10	9 … 2	1	0
1	0	0	0	1	0	index8	0	0
FUNCT6						index8>=8, index8<64	OP=C0	

NOTE For this encoding to decode as *TBLJ*, *index8>=8 && index8<64* otherwise it's a *different* encoding

Syntax

```
c.tblj #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.

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 the index passed to the C.TBLJ* in
the assembler

switch(XLEN) {
  32:  table_address[XLEN-1:0] = TBLJALVEC.base + index8<<2;
  64:  table_address[XLEN-1:0] = TBLJALVEC.base + index8<<3;
  128: table_address[XLEN-1:0] = TBLJALVEC.base + index8<<4;
}

//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];
    128: LQ 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
    JALR zero, target_address[XLEN-1:0]&~0x1;
  }
}
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.51)	0.51	Plan

c.tbljal

Synopsis

table jump and link to ra, 16-bit encoding

Mnemonic

c.tbljal *#index*

Encoding (RV32, RV64, RV128)

15	14	13	12	11	10	9–2	1	0
1	0	0	0	1	0	index8	0	0
FUNCT6						index8>=64	OP=C0	

NOTE For this encoding to decode as *TBLJAL*, *index8>=64* otherwise it's a *different* encoding

Syntax

```
c.tbljal #index
```

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.

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 the index passed to the C.TBLJ* in
the assembler

switch(XLEN) {
  32:  table_address[XLEN-1:0] = TBLJALVEC.base + index8<<2;
  64:  table_address[XLEN-1:0] = TBLJALVEC.base + index8<<3;
  128: table_address[XLEN-1:0] = TBLJALVEC.base + index8<<4;
}

//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];
    128: LQ 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
    JALR ra, target_address[XLEN-1:0]&~0x1;
  }
}
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.51)	0.51	Plan

c.tbljalm

Synopsis

table jump and link to t0, 16-bit encoding

Mnemonic

c.tbljalm *#index*

Encoding (RV32, RV64, RV128)

15					10	9 … 2	1	0
1	0	0	0	1	0	index8	0	0
FUNCT6						index8<8	OP=C0	

NOTE For this encoding to decode as *TBLJALM*, *index8<8* otherwise it's a *different* encoding

NOTE M in the mnemonic refers to millicode, as t0 is used for calling millicode routines.

Syntax

```
c.tbljalm #index
```

Description

This instruction is used to dereference a table of PCs, and then jumps to the dereferenced PC and links to t0.

For further information see Table Jump Instructions.

Prerequisites

The C-extension must also be configured.

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 the index passed to the C.TBLJ* in
the assembler

switch(XLEN) {
  32:  table_address[XLEN-1:0] = TBLJALVEC.base + index8<<2;
  64:  table_address[XLEN-1:0] = TBLJALVEC.base + index8<<3;
  128: table_address[XLEN-1:0] = TBLJALVEC.base + index8<<4;
}

//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];
    128: LQ 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
    JALR t0, target_address[XLEN-1:0]&~0x1;
  }
}
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.51)	0.51	Plan

TBLJALVEC CSR

Synopsis

Table jump base vector and control register

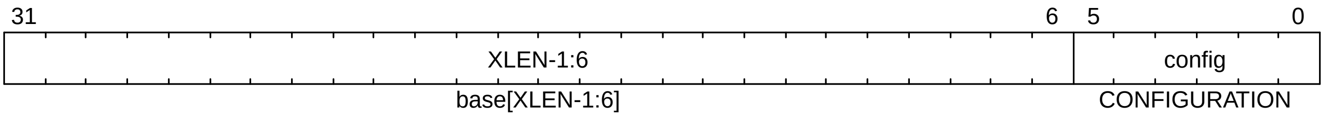
Address

TBD

Permissions

URW

Format (RV32, RV64, RV128)



Description

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

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

The memory pointed to by *TBLJALVEC.base* only requires eXecute permission. Read/Write access is not required once the jump table/vector table has been configured. If code is to be emulated then Read access is also required, but the table jump instructions themselves don't require this.

Table 9. TBLJALVEC.config definition

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

TBLJALVEC.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

TBLJALVEC adds architectural state to the context, therefore must be saved/restore on context switch.

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 0.51)	0.51	Plan

c.mva01s07

Synopsis

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

Mnemonic

c.mva01s07 *sreg1, sreg2*

Encoding (RV32, RV64, RV128)

15					10	9		7	6	5	4		2	1	0
0	1	0	0	1	1	1	sreg1		1	1		sreg2		0	1
		FUNCT6												OP=C1	

Syntax

```
c.mva01s07 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 10. 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 This instruction does not directly expand to a single 32-bit encoding.

Prerequisites

The C-extension must also be configured.

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 0.51)	0.51	Plan

c.mvs07a01

Synopsis

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

Mnemonic

c.mvas07a01 *sreg1*, *sreg2*

Encoding (RV32, RV64, RV128)

15					10	9		7	6	5	4		2	1	0
0	1	0	0	1	1	1	sreg1		1	1		sreg2		0	1
FUNCT6														OP=C1	

Syntax

```
c.mvas07a01 sreg1, sreg2
```

Description

This instruction moves *a0* into *sreg1* and *a1* into *sreg2*. The execution is atomic, so it is not possible to observe state where only one of *sreg1* or *sreg2* 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 This instruction does not directly expand to a single 32-bit encoding.

Prerequisites

The C-extension must also be configured.

Operation

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

if (sreg1==sreg2) {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[sreg1] = X[10]
X[sreg2] = X[11]
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.51)	0.51	Plan

PUSH/POP register instructions

These instructions are collectively referred to as PUSH/POP:

- c.push: push registers to stack memory, 16-bit encoding
- push: push registers to stack memory, 32-bit encoding
- c.popret: pop registers and return, 16-bit encoding
- popret: pop registers from the stack and return, 32-bit encoding
- c.pop: pop registers from the stack, 16-bit encoding
- pop: pop registers from the stack, 32-bit encoding

The term PUSH refers to both 16 and 32-bit encodings (C.PUSH, PUSH).

The term POP refers to both 16 and 32-bit encodings of POP (C.POP, POP).

The term POPRET refers to both 16 and 32-bit encodings of POPRET (C.POPRET, POPRET).

Common details for these instructions are in this section.

NOTE This version does not fully include *areg_list* and will need to be updated if it's included in *push/c.push*.

PUSH/POP overview

PUSH, POP, POPRET along with the 16-bit forms are used to reduce the size of function prologues and epilogues.

1. The PUSH instruction
   - pushes(stores) the registers specified in *reg_list* to the stack
   - if *areg_list* is included, moves the registers in the *areg_list* into *s* registers
     - *areg_list* is determined automatically from *rlist*, it cannot be arbitrarily specified. The definition is in c.push: push registers to stack memory, 16-bit encoding and push: push registers to stack memory, 32-bit encoding ;
   - adjusts the stack pointer by the *stack_adjustment*
2. The POP instruction
   - pops(loads) the registers in *reg_list* from the stack
   - if *ret_val* is included, moves the specified constant value into *a0* as the return value
   - adjusts the stack pointer by the *stack_adjustment*.
3. POPRET has the same behaviour as POP, followed by RET.

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

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

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

with PUSH/POPRET this reduces to

```
20405458 <function>:
20405458: <16-bit>            push   {ra,s0-s2},{a0-a2},-16
<function body>
20405496: <16-bit>            popret {ra,s0-s2},{0}, 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	The C.PUSH 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.PUSH/PUSH and will cost some performance.

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

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.

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.
- 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 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 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:

```
popret   {ra, s0-s3}, {1}, 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, 1
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.

Compiling PUSH/POP for size or performance

There are cases where there are choices about whether to select the 16-bit or 32-bit encoding. The 32-bit encodings offer a smaller stack adjustment range than using a 16-bit encoding and an additional C.ADDI16SP instruction. Therefore using the 32-bit encoding will not reduce the code size if the stack adjustment is out of range of the 16-bit encoding.

The main performance/code-size trade-offs are

- whether the register list available in the 16-bit encodings matches the required list, and so whether extra registers are included by the 16-bit encoding
- whether areg_list includes redundant moves

The recommendation is that the 32-bit encoding should be selected only if compiling for performance and either

- the register list is not available in the 16-bit encoding
- areg_list includes redundant moves

In addition, for POPRET, the 32-bit encoding allows more return values than the 16-bit encoding. Therefore the recommendation is that the 32-bit encoding should be selected if the 32-bit encoding allows the required return value.

Included in

Extension	Minimum version	Lifecycle state
Zces ([zces])	0.51	Plan

Table Jump Instructions

These instructions are collectively referred to as table jump:

- c.tblj: table jump without link, 16-bit encoding
- c.tbljal: table jump and link to ra, 16-bit encoding
- c.tbljalm: table jump and link to t0, 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.

This example uses GCC10 output:

```
00e084be <function>:
  #64-bit AUIPC/JALR sequence
  e084be:    001f8317           auipc t1,0x1f8
  e084c2:    18a302e7           jalr  t0,394(t1) # 1000648 <__riscv_save_0>

  e084c6:    86b2               mv    a3,a2
  e084c8:    862e               mv    a2,a1
  e084ca:    800005b7           lui   a1,0x80000
  e084ce:    fff5c593           not   a1,a1

  #32-bit JAL
  e084d2:    f61ff0ef           jal   ra,e08432 <function2>

  #64-bit AUIPC/JALR sequence
  e084d6:    001f8317           auipc t1,0x1f8
  e084da:    19630067           jr    406(t1) # 100066c <__riscv_restore_0>
```

using Table Jump we can reduce this as follows (accepting gaps in the PCs as code has been deleted)

```
00e084be <vsprintf>:
  e084be:   <16-bit>              tbljalm #x ;#<maps to __riscv_save_0>

  e084c6:   86b2                  mv      a3,a2
  e084c8:   862e                  mv      a2,a1
  e084ca:   800005b7              lui     a1,0x80000
  e084ce:   fff5c593              not     a1,a1

  e084d2:   <16-bit>              tbljal #y ;#<maps to function2>

  e084da:   <16-bit>              tblj    #z ;#<maps to __riscv_restore_0>
```

There is a single lookup table of up to 256 addresses for Table Jump, which is built by the linker. The linker then substitutes the code as shown in the example above where the 32-byte function is reduced to 18-bytes giving ~ 56% saving. Clearly the lookup table takes some space, but this is a minimal overhead for repeated functions such as the save/restore routines.

Table jump allows the linker to:

- replace 32-bit J calls with C.TBLJ
- replace 32-bit JAL ra calls with C.TBLJAL
- replace 32-bit JAL t0 calls with C.TBLJALM (M for Millicode)
- replace 64-bit AUIPC/JR calls to fixed locations with C.TBLJ
- replace 64-bit AUIPC/JALR ra calls to fixed locations with C.TBLJAL
- replace 64-bit AUIPC/JALR t0 calls to fixed locations with C.TBLJALM
    - 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 tbljalvec CSR, table jump base vector and control register), each table entry is XLEN bits.

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

- C.TBLJALM: entries 0-7, link to *t0*
- C.TBLJ : entries 8-63, link to *zero*
- C.TBLJAL : 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
JAL t0	E*2-(XLEN/8) bytes
AUIPC+JALR t0	F*6-(XLEN/8) bytes

Each function is called by using one of the three 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_tblj    = A * 2 + B * 6 - 2*(XLEN-8)
saving_per_function_c_tbljal  = C * 2 + D * 6 - 2*(XLEN-8)
saving_per_function_c_tbljalm = E * 2 + F * 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 seciton gives an overview of the behaviour, the exact operation is documented in the SAIL code for each instruction

- c.tbljalm SAIL code
- c.tbljal SAIL code
- c.tblj SAIL code

Included in

Extension
Minimum version
Lifecycle state
Zces ([zces])
0.51
Plan