Zces 0.53.4

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.

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

NOTE The PUSH/POP instructions share assembly mnemonics for different encodings. For further information see PUSH/POP Register Instructions.

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

- *reg_list* is a list containing 1 to 13 registers (ra and 0 to 12 s registers)
  - valid values: {ra}, {ra, s0}, {ra, s0-s1}, {ra, s0-s2}, ..., {ra, s0-s8}, {ra, s0-s9}, {ra, s0-s11}
  - note that {ra, s0-s10} is *not* valid, giving 12 lists not 13 for better encoding
- *areg_list* is a list containing 1 to 3 a registers
  - valid values: {a0}, {a0-a1}, {a0-a2}
- *stack_adj* is the total size of the stack frame.
  - valid values vary with register list length and the specific encoding, see the instruction pages for details.

RV32	RV64	Mnemonic	Instruction
✓	✓	c.push {reg_list}, -stack_adj	c.push: Create stack frame: push registers, allocate additional stack space.
✓	✓	c.push {reg_list}, {areg_list}, -stack_adj	c.pusha: Create stack frame: push registers, move A to S registers, allocate additional stack space.
✓	✓	c.pop {reg_list}, stack_adj	c.pop: Destroy stack frame: pop registers, deallocate stack frame.
✓	✓	c.popret {reg_list}, stack_adj	c.popret: Destroy stack frame: pop registers, deallocate stack frame, return.
✓	✓	c.popretz {reg_list}, stack_adj	c.popretz: Destroy stack frame: pop registers, deallocate stack frame, return zero.
✓	✓	c.jt #index	c.jt: jump via table without link
✓	✓	c.jalt #index	c.jalt: jump via table and link to ra
✓	✓	c.mva01s sreg1, sreg2	c.mva01s: move two s0-s7 registers into a0-a1

c.push

Synopsis

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

Mnemonic

c.push

Encoding (RV32, RV64)

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

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

Assembly Syntax

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

The variables used in the assembly syntax are defined below.

```
RV32E:

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

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: take_illegal_instruction_exception();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32E:

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

```
RV32I:

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

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

```
RV64:

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

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

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

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

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation

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

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

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

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

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

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

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

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

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

sp-=stack_adj;
```

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

```
addi sp, sp, -64;
sw  s2, 60(sp);
sw  s1, 56(sp);
sw  s0, 52(sp);
sw  ra, 48(sp);
```

RV32I Assembly example

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

Encoding: *rlist*=6, *spimm*=1

The equivalent interrupt safe assembly sequence is:

```
addi sp, sp, -32;
sw  s1, 28(sp);
sw  s0, 24(sp);
sw  ra, 20(sp);
```

RV32I Assembly example

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

Encoding: *rlist*=8, *spimm*=2

The equivalent interrupt safe assembly sequence is:

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

RV32I Assembly example

```
c.push {ra, s0-s11}, -112
```

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

The equivalent interrupt safe assembly sequence is:

```
addi sp, sp, 112;
sw   s11, 108(sp);
sw   s10, 104(sp);
sw   s9,  100(sp);
sw   s8,   96(sp);
sw   s7,   92(sp);
sw   s6,   88(sp);
sw   s5,   84(sp);
sw   s4,   80(sp);
sw   s3,   76(sp);
sw   s2,   72(sp);
sw   s1,   68(sp);
sw   s0,   64(sp);
sw   ra,   60(sp);
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.4)	0.53.4	Stable

c.pusha

Synopsis

Create stack frame: store ra and 1 to 12 saved registers to the stack frame, move arguments into saved registers, optionally allocate additional stack space.

Mnemonic

c.pusha

Encoding (RV32, RV64)

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

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

NOTE *rlist* value 4 is reserved, unlike for *c.push*, *c.pop*, *c.popret*, *c.popretz*

Assembly Syntax

```
c.push {reg_list},  {areg_list},  -stack_adj
c.push {xreg_list}, {xareg_list}, -stack_adj
```

The variables used in the assembly syntax are defined below.

```
RV32E:

switch (rlist){
  case  5: {reg_list="ra, s0";     areg_list="a0";    xareg_list="x10";}
  case  6: {reg_list="ra, s0-s1";  areg_list="a0-a1"; xareg_list="x10-x11";}
  default: take_illegal_instruction_exception();
}
```

```
RV32I, RV64:

switch (rlist){
  case     5: {reg_list="ra, s0";     areg_list="a0";    xareg_list="x10";}
  case     6: {reg_list="ra, s0-s1";  areg_list="a0-a1"; xareg_list="x10-x11";}
  case 7..15: {reg_list="ra, s0-s2";  areg_list="a0-a2"; xareg_list="x10-x12";}
  default:    take_illegal_instruction_exception();
}
```

```
RV32E:

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

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: take_illegal_instruction_exception();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32E:

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

```
RV32I:

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

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

```
RV64:

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

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pushes (stores) the registers in *reg_list* to stack memory, moves *areg_list* into correspondingly 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.

Stack Adjustment Calculation

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

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

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

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

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

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

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

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

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

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_adj;
```

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

```
addi sp, sp, -64;
sw  s2, 60(sp);
sw  s1, 56(sp);
sw  s0, 52(sp);
sw  ra, 48(sp);
mv  s0, a0;
mv  s1, a1;
mv  s2, a2;
```

RV32I Assembly example

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

Encoding: *rlist*=6, *spimm*=1

The equivalent interrupt safe assembly sequence is:

```
addi sp, sp, -32;
sw  s1, 28(sp);
sw  s0, 24(sp);
sw  ra, 20(sp);
mv  s0, a0;
mv  s1, a1;
```

RV32I Assembly example

```
c.push  {ra, s0-s3}, {a0-a2}, -64
```

Encoding: *rlist*=8, *spimm*=2

The equivalent interrupt safe assembly sequence is:

```
addi sp, sp, 64;
sw  s3, 60(sp);
sw  s2, 56(sp);
sw  s1, 52(sp);
sw  s0, 48(sp);
sw  ra, 44(sp);
mv  s0, a0;
mv  s1, a1;
mv  s2, a2;
```

RV32I Assembly example

```
c.push {ra, s0-s11}, {a0-a2}, -112
```

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

The equivalent interrupt safe assembly sequence is:

```
addi sp, sp, 112;
sw   s11, 108(sp);
sw   s10, 104(sp);
sw   s9,  100(sp);
sw   s8,   96(sp);
sw   s7,   92(sp);
sw   s6,   88(sp);
sw   s5,   84(sp);
sw   s4,   80(sp);
sw   s3,   76(sp);
sw   s2,   72(sp);
sw   s1,   68(sp);
sw   s0,   64(sp);
sw   ra,   60(sp);
mv   s0, a0;
mv   s1, a1;
mv   s2, a2;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.4)	0.53.4	Stable

c.pop

Synopsis

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

Mnemonic

c.pop

Encoding (RV32, RV64)

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

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

Assembly Syntax

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

The variables used in the assembly syntax are defined below.

```
RV32E:

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

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: take_illegal_instruction_exception();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32E:

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

```
RV32I:

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

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

```
RV64:

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

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

This instruction pop (loads) the registers in *reg_list* 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.

Stack Adjustment Calculation

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

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

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

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

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

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

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

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

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

sp+=stack_adj;
```

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

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

RV32I Assembly example

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

Encoding: *rlist*=7, *spimm*=2

The equivalent interrupt safe assembly sequence is:

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

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

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

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

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

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.4)	0.53.4	Stable

c.popret

Synopsis

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

Mnemonic

c.popret

Encoding (RV32, RV64)

15	14	13	12	11	10	9	8	7–4	3–2	1	0
1	0	1	1	1	1	1	0	rlist	spimm[5:4]	1	0

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

Assembly Syntax

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

The variables used in the assembly syntax are defined below.

```
RV32E:

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

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: take_illegal_instruction_exception();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32E:

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

```
RV32I:

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

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

```
RV64:

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

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

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

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

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation

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

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

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

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

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

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

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

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

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

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

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

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

RV32I Assembly example

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

Encoding: *rlist*=7, *spimm*=2

The equivalent interrupt safe assembly sequence is:

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

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

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

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

```
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 0.53.4)	0.53.4	Stable

c.popretz

Synopsis

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

Mnemonic

c.popretz

Encoding (RV32, RV64)

15	14	13	12	11	10	9	8	7–4	3–2	1	0
1	0	1	1	1	1	0	0	rlist	spimm[5:4]	1	0

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

Assembly Syntax

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

```
RV32E:

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

```
RV32I, RV64:

switch (rlist){
  case  4: {reg_list="ra";         xreg_list="x1";}
  case  5: {reg_list="ra, s0";     xreg_list="x1, x8";}
  case  6: {reg_list="ra, s0-s1";  xreg_list="x1, x8-x9";}
  case  7: {reg_list="ra, s0-s2";  xreg_list="x1, x8-x9, x18";}
  case  8: {reg_list="ra, s0-s3";  xreg_list="x1, x8-x9, x18-x19";}
  case  9: {reg_list="ra, s0-s4";  xreg_list="x1, x8-x9, x18-x20";}
  case 10: {reg_list="ra, s0-s5";  xreg_list="x1, x8-x9, x18-x21";}
  case 11: {reg_list="ra, s0-s6";  xreg_list="x1, x8-x9, x18-x22";}
  case 12: {reg_list="ra, s0-s7";  xreg_list="x1, x8-x9, x18-x23";}
  case 13: {reg_list="ra, s0-s8";  xreg_list="x1, x8-x9, x18-x24";}
  case 14: {reg_list="ra, s0-s9";  xreg_list="x1, x8-x9, x18-x25";}
  //note - to include s10, s11 must also be included
  case 15: {reg_list="ra, s0-s11"; xreg_list="x1, x8-x9, x18-x27";}
  default: take_illegal_instruction_exception();
}
stack_adj      = stack_adj_base + spimm[5:4] * 16;
```

```
RV32E:

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

```
RV32I:

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

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

```
RV64:

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

Valid values:
switch (rlist) {
  case  4.. 5: stack_adj = [ 16| 32| 48| 64];
  case  6.. 7: stack_adj = [ 32| 48| 64| 80];
  case  8.. 9: stack_adj = [ 48| 64| 80| 96];
  case 10..11: stack_adj = [ 64| 80| 96|112];
  case 12..13: stack_adj = [ 80| 96|112|128];
  case     14: stack_adj = [ 96|112|128|144];
  case     15: stack_adj = [112|128|144|160];
}
```

Description

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

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

For further information see PUSH/POP Register Instructions.

Stack Adjustment Calculation

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

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

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

Prerequisites

The C-extension must also be configured.

32-bit equivalent

No direct equivalent encoding exists

Operation

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

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

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

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

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

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

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

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

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

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

RV32I Assembly example

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

Encoding: *rlist*=7, *spimm*=2

The equivalent interrupt safe assembly sequence is:

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

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

```
lw   s3, 44(sp);
lw   s2, 40(sp);
lw   s1, 36(sp);
lw   s0, 32(sp);
lw   ra, 28(sp);
li   a0, 0;
addi sp, sp, 48;
ret;
```

RV32I Assembly example

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

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

The equivalent interrupt safe assembly sequence is:

```
lw   s4, 60(sp);
lw   s3, 56(sp);
lw   s2, 52(sp);
lw   s1, 48(sp);
lw   s0, 44(sp);
lw   ra, 40(sp);
li   a0, 0;
addi sp, sp, 64;
ret;
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.4)	0.53.4	Stable

c.jt

Synopsis

jump via table without link

Mnemonic

c.jt

Encoding (RV32, RV64)

15		13	12		10	9	2	1	0
1	0	1	0	0	1	index8		1	0

NOTE For this encoding to decode as *c.jt*, *index8<64*, otherwise it decodes as c.jalt: jump via table and link to ra.

Assembly 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.JT/C.JALT
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 0.53.4)	0.53.4	Stable

c.jalt

Synopsis

jump via table and link to ra

Mnemonic

c.jalt

Encoding (RV32, RV64)

15		13	12		10	9	2	1	0
1	0	1	0	0	1	index8		1	0

NOTE For this encoding to decode as *c.jalt*, *index8>=64*, otherwise it decodes as c.jt: jump via table without link.

Assembly 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.JT/C.JALT
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 0.53.4)	0.53.4	Stable

JVT CSR

Synopsis

Table jump base vector and control register

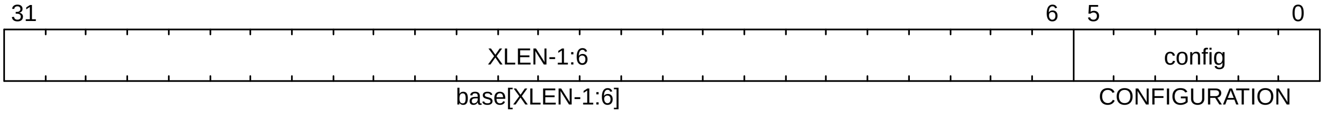
Address

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 1. 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 0.53.4)	0.53.4	Stable

c.mva01s

Synopsis

Move two s0-s7 registers into a0-a1

Mnemonic

c.mva01s

Encoding (RV32, RV64)

15	14	13	12	11	10	9–7	6	5	4–2	1	0
1	0	1	0	1	1	sreg1	1	1	sreg2	1	0

Assembly 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.

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

NOTE The *s* register mapping is taken from the UABI, and may not match the currently unratified EABI. *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.

if (RV32E && (sreg1>1 || sreg2>1)) {
  take_illegal_instruction_exception();
}

xreg1 = {sreg1[2:1]>0,sreg1[2:1]==0,sreg1[2:0]};
xreg2 = {sreg2[2:1]>0,sreg2[2:1]==0,sreg2[2:0]};

X[10] = X[xreg1];
X[11] = X[xreg2];
```

Included in

Extension	Minimum version	Lifecycle state
Zces (Zces 0.53.4)	0.53.4	Stable

PUSH/POP register instructions

These instructions are collectively referred to as PUSH/POP:

- c.push: Create stack frame: push registers, allocate additional stack space.
- c.pusha: Create stack frame: push registers, move A to S registers, allocate additional stack space.
- c.pop: Destroy stack frame: pop registers, deallocate stack frame.
- c.popret: Destroy stack frame: pop registers, deallocate stack frame, return.
- c.popretz: Destroy stack frame: pop registers, deallocate stack frame, return zero.

The term PUSH refers to *c.push and c.pusha*. 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 and c.popretz.*

Common details for these instructions are in this section.

The difference between *c.push* and *c.pusha* is whether argument registers are moved into saved registers. The same mnemonic is used and the difference is in the argument list. For example:

```
c.push {ra,s0-s3},          -32;# maps to c.push
c.push {ra,s0-s3}, {a0-a3}, -32;# maps to c.pusha
```

It is specified this way so the list of argument registers is explicit in the syntax, otherwise it would be unclear which registers were moved.

PUSH/POP functional 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 frame
   - *c.pusha* also moves the registers in *areg_list* into *s* registers
     - In order to save encoding space *areg_list* is determined automatically from the register list and cannot be arbitrarily specified.
     - moving argument registers into saved registers is to save them before setting up the arguments before calling the next function
   - adjust the stack pointer to create the stack frame
2. The POP instruction
   - pops(loads) the registers in the register list from the stack frame
   - adjusts the stack pointer to destroy the stack frame
3. The POPRET instructions
   - pop(load) the registers in the register list from the stack from
   - *c.popretz* also moves zero into into *a0* as the return value
   - adjust the stack pointer to destroy the stack frame
   - execute a *ret* instruction to return from the function

Example usage

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

```
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.

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

register list handling

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

areg_list handling

c.pusha includes *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
  e01128:       ce06                    sw      ra,28(sp) ; #c.push
  e0112a:       cc22                    sw      s0,24(sp) ; #c.push
  e0112c:       ca26                    sw      s1,20(sp) ; #c.push
  e0112e:       c84a                    sw      s2,16(sp) ; #c.push
  e01130:       c64e                    sw      s3,12(sp) ; #c.push
  e01132:       c452                    sw      s4,8(sp)  ; #c.push
  e01134:       c256                    sw      s5,4(sp)  ; #c.push
  e01136:       c05a                    sw      s6,0(sp)  ; #c.push
  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* 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* 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* 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
  - xTVAL 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)
  - xTVAL 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
  - xTVAL 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
  - xTVAL 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 xTVAL may be hardwired to zero in an implementation. Recovering from faults such as page faults requires that it is implemented.

In all cases MEPC contains 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 must be 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 when execution resumes.

The stack pointer adjustment must only be committed once it is certain that all of the stores will complete without triggering any precise faults (for example, page faults). Stores may also return imprecise faults from the bus. It is platform defined whether the core implementation waits for the bus responses before continuing to the final stage of the sequence, or handles errors responses after completing the PUSH instruction.

For example:

```
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
# therefore these updates may be visible in the interrupt/exception handler below
the stack pointer
sw  s5, -4(sp);
sw  s4, -8(sp);
sw  s3,-12(sp);
sw  s2,-16(sp);
sw  s1,-20(sp);
sw  s0,-24(sp);
sw  ra,-28(sp);

# these must only execute once, and will only execute after all stores completed
without any precise faults
# all instructions must execute atomically
# therefore these updates are not visible in the interrupt/exception handler
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
# therefore these updates may be visible in the interrupt/exception handler
lw   s3, 28(sp);
lw   s2, 24(sp);
lw   s1, 20(sp);
lw   s0, 16(sp);
lw   ra, 12(sp);

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

Forward progress guarantee

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

Non-idempotent memory handling

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

Error detection

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

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 decreasing address order.

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 0.53.4)	0.53.4	Stable

Table Jump Instructions

These instructions are collectively referred to as table jump:

- c.jt: jump via table without link
- c.jalt: jump via table and link to ra

Common details for these instructions are in this section.

Table Jump Overview

Table jump is a form of dictionary compression used to reduce the code size of JAL / AUIPC+JALR / JR / AUIPC+JR instructions.

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

Table jump allows the linker to:

- replace 32-bit *j* calls with *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.

JVT

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

The table entry number is from the *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 2. 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. There are no data accesses involved in the execution of table jumps.

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 table jump instruction and the address in the table map to different virtual memory pages or PMP regions
- watchpoint trigger or debug halt caused by a trigger
- external debug halt
  - the halt can treat the whole sequence atomically, or interrupt mid sequence (implementation defined)
- interrupts
  - these may arrive at any time. An implementation can choose whether to interrupt the sequence or not.

For exceptions and interrupts MEPC contain the PC of the table jump instruction, MCAUSE is set as expected for the type of fault and MTVAL contains the address which caused the 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 0.53.4)
0.53.4
Stable