RISC-V Push/Pop & Frame Pointer Proposal

Version 0.2

James Ball
jameball@qti.qualcomm.com
Overview

Problem
• Compiler can’t use Zc push/pop instructions when frame pointer requested
• Results in significant code size impact due to omission of push/pop instructions

What is a Frame Pointer (FP)?
• Optional software managed linked list of function call frames using the FP register (x8 AKA s0)

History
• Push/pop stack order consistent with future potential load/store multiple instructions rather than ABI order
  • See GitHub Issue 194 and Code Size TG post
• ABI 1.1 specifies portion of stack order when FP present
  • Not yet ratified but already implemented across all toolchains
  • Too late to change ABI 1.1 and toolchains (could break backwards-compatibility)

Motivation for Push/Pop and FP
• FP allows stack backtrace without requiring access to debug information in ELF (not always available)
• Can’t find return address (variable offset relative to current SP) or caller’s stack frame without FP
• Example scenarios
  • Program detects a fatal condition and wants to print stack track before exiting
  • Debugging on hardware when ELF debug information isn’t present
Background
Frame Pointer 101

Enabled by compiler option
- \(-fno-omit-frame-pointer\)

Creates linked list of stack frames
- Current FP points to head

ABI 1.1 defines fixed FP relationships
- FP → SP when called
- FP-1*XLEN/8 → Return Address
- FP-2*XLEN/8 → Caller’s FP

Function Call Order

Stack

Function A Stack Frame
- Return Address
- Caller’s FP (Null)
- Other Saved Registers and Locals

Function B Stack Frame
- Return Address
- Caller’s FP
- Other Saved Registers and Locals

Function C Stack Frame
- Return Address
- Caller’s FP
- Other Saved Registers and Locals

Decreasing Memory Addresses
- Current FP
- Current SP (Top of stack)
RISC-V ABI vs. Zc Push/Pop Stack

Zc push/pop instructions store ra & s0 opposite order to RISC-V ABI 1.1

Implication: Compiler can't use push/pop when frame pointer requested

<table>
<thead>
<tr>
<th>Register</th>
<th>ABI Name</th>
<th>Description</th>
<th>Saver</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0</td>
<td>zero</td>
<td>Hard-wired zero</td>
<td>—</td>
</tr>
<tr>
<td>x1</td>
<td>ra</td>
<td>Return address</td>
<td>Caller</td>
</tr>
<tr>
<td>x2</td>
<td>sp</td>
<td>Stack pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x3</td>
<td>gp</td>
<td>Global pointer</td>
<td>—</td>
</tr>
<tr>
<td>x4</td>
<td>tp</td>
<td>Thread pointer</td>
<td>—</td>
</tr>
<tr>
<td>x5</td>
<td>t0</td>
<td>Temporary/alternate link register</td>
<td>Caller</td>
</tr>
<tr>
<td>x6-7</td>
<td>t1-2</td>
<td>Temporaries</td>
<td>Callee</td>
</tr>
<tr>
<td>x8</td>
<td>s0/fp</td>
<td>Saved register/frame pointer</td>
<td>Callee</td>
</tr>
<tr>
<td>x9</td>
<td>a1</td>
<td>Saved register</td>
<td>Callee</td>
</tr>
<tr>
<td>x10-11</td>
<td>a0-1</td>
<td>Function arguments/return values</td>
<td>Caller</td>
</tr>
<tr>
<td>x12-17</td>
<td>a2-7</td>
<td>Function arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>x18-27</td>
<td>a2-11</td>
<td>Saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>x28-31</td>
<td>t3-6</td>
<td>Temporaries</td>
<td>Callee</td>
</tr>
<tr>
<td>f0-7</td>
<td>ft0-7</td>
<td>FP temporaries</td>
<td>Caller</td>
</tr>
<tr>
<td>f8-9</td>
<td>fa0-1</td>
<td>FP saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>f10-11</td>
<td>fa0-1</td>
<td>FP arguments/return values</td>
<td>Caller</td>
</tr>
<tr>
<td>f12-17</td>
<td>fa2-7</td>
<td>FP arguments</td>
<td>Caller</td>
</tr>
<tr>
<td>f18-27</td>
<td>fa2-11</td>
<td>FP saved registers</td>
<td>Callee</td>
</tr>
<tr>
<td>f28-31</td>
<td>ft8-11</td>
<td>FP temporaries</td>
<td>Caller</td>
</tr>
</tbody>
</table>
Zc Push/Pop Instruction Equivalents

Examples assume XLEN=32

```
cm.push {ra, s0-s2},-16 equivalent to:
sw     s2,   -4(sp)
sw     s1,   -8(sp)
sw     s0,  -12(sp)
sw     ra,  -16(sp)
addi   sp, sp, -16

Compiler with FP
sw     ra,   -4(sp)
sw     s0,   -8(sp)  # s0 is fp
sw     s1,   -12(sp)
sw     s2,  -16(sp)
mov     fp, sp     # Save initial sp in fp
addi   sp, sp, -16
```

Extra instruction to manage fp

Order reversed from push/pop
Solutions
Constraints on Solutions

Don’t break ecosystem that uses the FP
- Stack backtrace utility function breaks if ABI 1.1 layout changed to match push/pop
- Code pre-compiled in library couldn’t even have #ifdef added into source for additional layout option
- Could in theory have other software that relies on stack layout

Don’t break ecosystem not using FP
- Changing order of existing push/pop instructions could break software in theory
- Code could do load/store to stack contents added by push instruction
- Not sure this use case needs support
- Debugger might rely on existing stack layout
- Hopefully, DWARF debug information in ELF describes layout and debugger uses this
- Order can’t be controlled by a new CSR bit since DWARF specifies layout statically

Don’t create a burden for RISC-V community
- Qualcomm needs a standard solution with compiler & tools support that can be upstreamed
- Non-standard solution creates ongoing burden of adding support to RISC-V GCC/LLVM on every release
Two Potential Solutions

1. Create new HW extension to change push/pop order to match ABI
   - Presence of new extension in an implementation means order always changed

2. Create new push/pop instructions to match ABI
   - Uses additional opcode space

\[ \text{cm.push} \]
Just reverse order of register numbers (e.g., 1, 8, 9, ..., 26, 27)

\[ \text{cm.pop/popret/popretz} \]
Just reverse order of register numbers like in cm.push
Addition to Proposed Solution - New cm.pushfp instruction

Eliminates code size penalty of including FP

Same behavior as cm.push except saves initial sp value to fp
  • Same arguments as cm.push but only used when FP present
  • Don’t need cm.popfp since cm.pop instructions can already pop FP (x8 AKA s0)
  • Qualcomm is patenting cm.pushfp and will donate patent to RVI

Is cm.pushfp worth adding?
  • Uses 6 bits of 16-bit opcode space
  • Saves 2 bytes in every function’s prologue
  • Other Zcmp instructions with same cost offer lower code size reduction
    • For example, cm.popretz saves 2 bytes only when function returns zero

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>09</th>
<th>08</th>
<th>07</th>
<th>06</th>
<th>05</th>
<th>04</th>
<th>03</th>
<th>02</th>
<th>01</th>
<th>00</th>
<th>opcode</th>
<th>Description</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM.MVSA01</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>r15</td>
<td>0</td>
<td>1</td>
<td>r2s'</td>
<td>1</td>
<td>0</td>
<td>Move 2 registers: r15 = a0; r2s' = a1</td>
<td>Matching encoding Zcmp CM.MVSA01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM.MVA01S</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>r15</td>
<td>1</td>
<td>1</td>
<td>r2s'</td>
<td>1</td>
<td>0</td>
<td>Move 2 registers: a0 = r15; a1 = r2s'</td>
<td>Matching encoding Zcmp CM.MVA01S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM.PUSH</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>urlist</td>
<td>[5:0]</td>
<td>1</td>
<td>0</td>
<td>Function entrance (prologue), including stack allocation</td>
<td>Matching encoding Zcmp CM.PUSH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM.PUSHFP</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>urlist</td>
<td>[5:0]</td>
<td>1</td>
<td>0</td>
<td>Function entrance (prologue), including stack allocation and frame pointer</td>
<td>New proposed instruction</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM.POP</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>urlist</td>
<td>[5:0]</td>
<td>1</td>
<td>0</td>
<td>Function epilogue, including stack deallocation</td>
<td>Matching encoding Zcmp CM.POP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM.POPRETZ</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>urlist</td>
<td>[5:0]</td>
<td>1</td>
<td>0</td>
<td>Function exit with return zero (epilogue), including stack deallocation, a0 = 0</td>
<td>Matching encoding Zcmp CM.POPRETZ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM.POPRET</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>urlist</td>
<td>[5:0]</td>
<td>1</td>
<td>0</td>
<td>Function exit (epilogue), including stack deallocation</td>
<td>Matching encoding Zcmp CM.POPRET</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Backup
What RISC-V ABI 1.1 says about Frame Pointer


Code that uses a frame pointer will construct a linked list of stack frames, where each frame links to its caller using a "frame record". A frame record consists of two XLEN values on the stack; the return address and the link to the next frame record. The frame pointer register will point to the innermost frame, thereby starting the linked list. By convention, the lowest XLEN value shall point to the previous frame, while the next XLEN value shall be the return address. The end of the frame record chain is indicated by the address zero appearing as the next link in the chain.

After the prologue, the frame pointer register will point to the Canonical Frame Address or CFA, which is the stack pointer value on entry to the current procedure. The previous frame pointer and return address pair will reside just prior to the current stack address held in $fp$. This puts the return address at $fp - \text{XLEN/8}$, and the previous frame pointer at $fp - 2 \times \text{XLEN/8}$. 