May 2024

Qualcomm

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

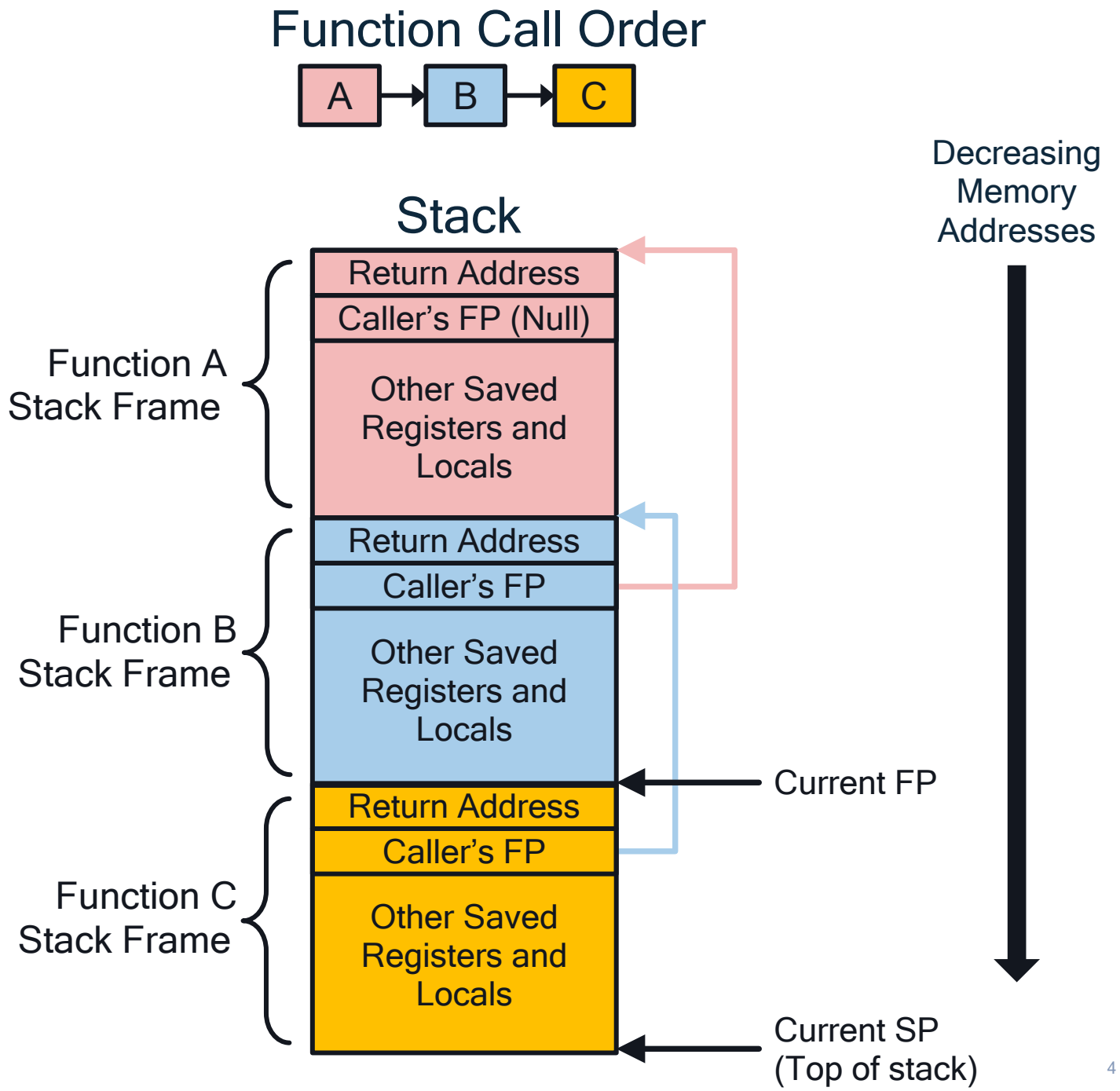
- -fno-omit-frame-pointer

Creates linked list of stack frames

- Current FP points to head

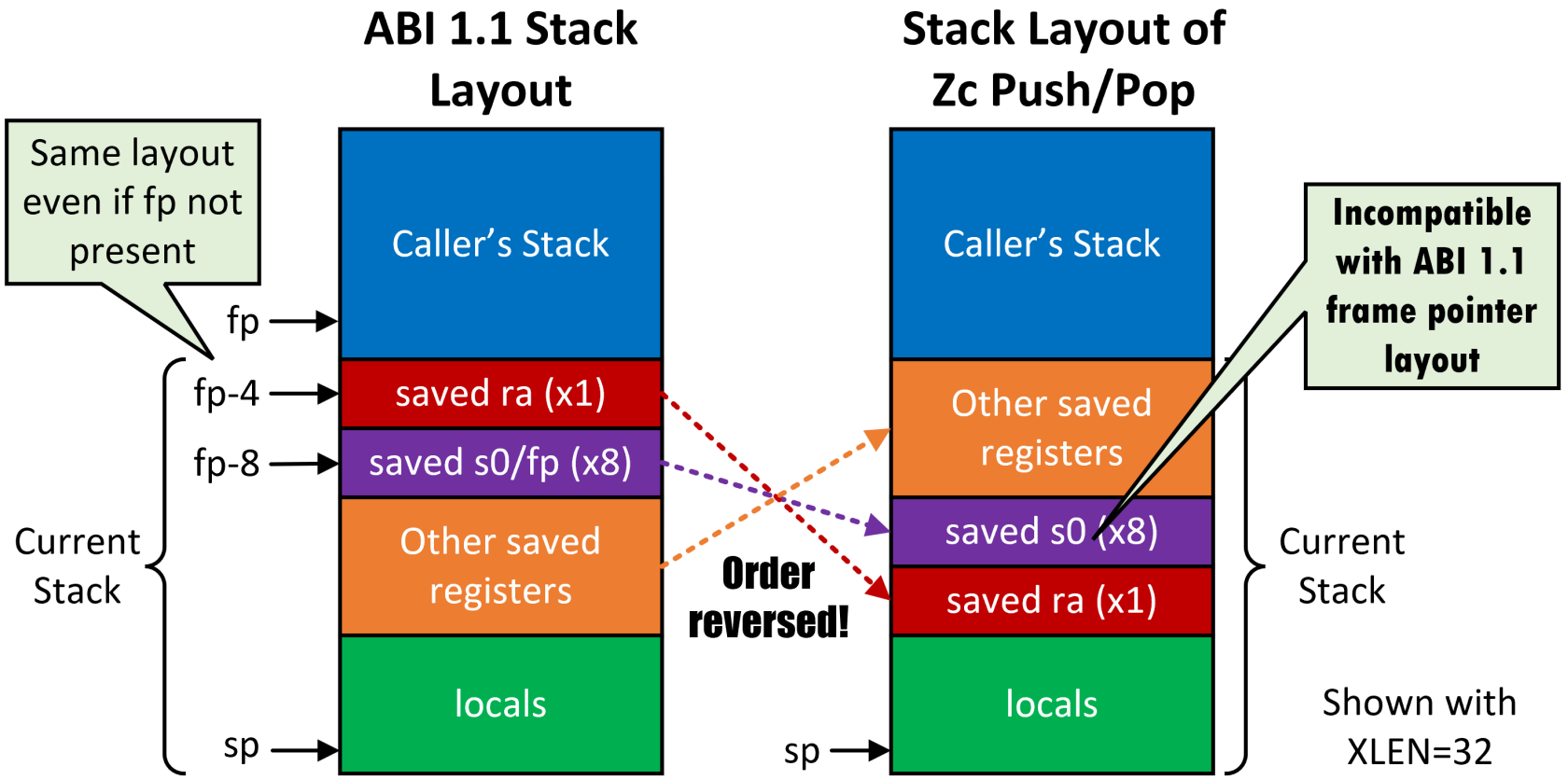
ABI 1.1 defines fixed FP relationships

- FP → SP when called
- FP-1*XLEN/8 → Return Address
- FP-2*XLEN/8 → Caller's FP



RISC-V ABI vs. Zc Push/Pop Stack

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



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

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

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
```

Order reversed from push/pop

Extra instruction to manage fp

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 assumes mem[fp-4] = “saved ra” and mem[fp-8] = “saved fp”
  - 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

cm.push

Just reverse order of register numbers (e.g., 1, 8, 9, …, 26, 27)

```
if (XLEN==32) bytes=4; else bytes=8;

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

cm.pop/popret/popretz

Just reverse order of register numbers like in cm.push

```
if (XLEN==32) bytes=4; else bytes=8;

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

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

Mnemonic	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00	Description	Comment
7	func6						r1s'			func2		r2s'			opcode			CMMV-type
CM.MVSA01	1	0	1	0	1	1	r1s'			0	1	r2s'			1	0	Move 2 registers: r1s' = a0; r2s' = a1	Matching encoding Zcmp CM.MVSA01
CM.MVA01S	1	0	1	0	1	1	r1s'			1	1	r2s'			1	0	Move 2 registers: a0 = r1s'; a1 = r2s'	Matching encoding Zcmp CM.MVA01S
	func6						func2		urlist				spimm		opcode			CMPP-type
CM.PUSH	1	0	1	1	1	0	0	0	urlist				[5:4]		1	0	Function entrance (prologue), including stack allocation	Matching encoding Zcmp CM.PUSH
CM.PUSHFP	1	0	1	1	1	0	0	1	urlist				[5:4]		1	0	Function entrance (prologue), including stack allocation and frame pointer	New proposed instruction
CM.POP	1	0	1	1	1	0	1	0	urlist				[5:4]		1	0	Function epilogue , including stack deallocation	Matching encoding Zcmp CM.POP
CM.POPRETZ	1	0	1	1	1	1	0	0	urlist				[5:4]		1	0	Function exit with return zero (epilogue) , including stack deallocation, a0 = 0	Matching encoding Zcmp CM.POPRET
CM.POPRET	1	0	1	1	1	1	1	0	urlist				[5:4]		1	0	Function exit (epilogue) , including stack deallocation	Matching encoding Zcmp CM.POPRETZ

Backup

What RISC-V ABI 1.1 says about Frame Pointer

From https://github.com/riscv-non-isa/riscv-elf-psabi-doc/blob/master/riscv-cc.adoc

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 - XLEN/8`, and the previous frame pointer at `fp - 2 * XLEN/8`.