Stack frames, how hard can it be?

dram

2023-04-19

About me

- `@dramforever` on most random platforms
- `https://dram.page`
- Call me ‘dram’

Basics

Calling convention

At function entry:

- ra (x1) return address
- sp (x2) stack pointer

At function exit:

- Restore sp
- Return value in `a0`
- Jump to original ra

Stack grows down

Just a random function

```c
long answer(void) {
    return 42;
}
```

Just a random function

```
    .section .text

    .global answer
    .type answer, @function

answer:
    li a0, 42
    ret

    .size answer, . - answer
```

Leaf functions

Leaf functions...

- Don’t have to use the stack
- (But can if they need to)
- No need to spill ra on the stack

More complex functions

```
int foo();

int bar() {
    return foo() + foo();
}
```

More complex functions

```
bar:
    addi sp, sp, -16
    sd ra, 8(sp)
    sd s0, 0(sp)

    call foo
    mv s0, a0         # s0 = foo()

    call foo
    addw a0, a0, s0   # a0 = s0 + foo()

    ld ra, 8(sp)
    ld s0, 0(sp)
    addi sp, sp, 16
    ret
```

More complex functions

At function entry (i.e. prologue):

- Decrement `sp`
- Save `ra` and used callee-saved regs on stack

At function exit (i.e. epilogue):

- Restore `ra` and callee-saved regs from stack
- Increment `sp`
- Return to `ra`

Wait, is ra callee-saved?

- Calling convention spec says no! `ra` is not preserved across calls
- `jal ra, ...` and `jalr ra, ...` destroys `ra` anyway

Wait, is ra callee-saved?

Technically allowed

```
ld t2, 8(sp)      # Load return address
jr t2
```

- Save `ra` anyway for non-leaf functions, because we need to know where to return to
- Restore `ra` anyway for return, because `jr ra` is recognized as a return for RAS prediction

Bonus: What do the insns look like?

- No stack usage, no stack frame allocation needed
- Small stack frame (< 512 bytes): `c.addi16sp`
- Larger frame (< 2048 bytes): `addi`

Bonus: What do the insns look like?

- No stack usage, no stack frame allocation needed
- Small stack frame (< 512 bytes): `c.addi16sp`
- Larger frame (< 2048 bytes): `addi`
- Very large frame (< 2 GiB):

```
li t0, -1000000 # lui/addi
sub sp, sp, t0
```

Bonus: What do the insns look like?

- Large-ish frames (< ~ 6000 bytes?):

```
# LLVM
addi    sp, sp, -2032
sd      ra, 2024(sp)
addi    sp, sp, -2048
addi    sp, sp, -944
```

If advantageous, may generate multiple `addi` instructions, bump sp in multiple steps

Frame pointer

Frame pointer

On RISC-V, x8/s0/fp is the frame pointer.

Standardized in April 2023 (!), frame pointer convention[1]:

- After function prologue, fp equals the CFA (canonical frame address), which is sp at function entry
- RV64: Previous fp at -16(fp), saved ra at -8(fp)
- (RV32: Previous fp at -8(fp), saved ra at -4(fp))

```
addi sp, sp, - N       # c.addi16sp
sd ra, (N - 8)(sp)     # c.sdsp
sd s0, (N - 16)(sp)    # c.sdsp
addi s0, sp, N         # c.addi4spn
```

[1] https://github.com/riscv-non-isa/riscv-elf-psabi-doc/pull/369

Frame pointer

Both GCC and LLVM force a frame pointer if the function uses `alloca` or VLA:

```c
void bar(size_t n) {
    char buf[n]; // Variable-length array
    foo(buf);
}
```

Force enabling fp

- gcc and clang: `-fno-omit-frame-pointer`
- rustc: `-Cforce-frame-pointers=yes`

Differences in GCC and LLVM

For leaf functions, GCC (as of 12.2.0) does not save `s0` and `ra` in the ‘correct’ place even with `-fno-omit-frame-pointer`[2]. LLVM does.

(They both do correctly have `fp = original sp`.)

[2] https://godbolt.org/z/vK7TTqTae

Differences in GCC and LLVM

```
    # GCC 12.2.0
answer:
    addi  sp, sp, -16
    sd    s0, 8(sp)
    addi  s0, sp, 16
    # ...
```

```
    # Clang 16.0.0
answer:
    addi  sp, sp, -16
    sd    ra, 8(sp)
    sd    s0, 0(sp)
    addi  s0, sp, 16
    # ...
```

Code size optimization

Code size optimization

Prologues and epilogues take up code size

- `__riscv_{save,restore}_N` out-of-line prologue and epilogue
- `Zcmp` extension

__riscv_{save,restore}_N

```c
int bar() { return foo() + foo(); }
```

`gcc -O -msave-restore` gives:

```
bar:
    call    t0,__riscv_save_1
    call    foo
    mv      s0,a0
    call    foo
    addw    a0,s0,a0
    tail    __riscv_restore_1
```

__riscv_{save,restore}_N

- __riscv_save_N: Push the first N callee-saved registers and ra, returns to t0
- __riscv_restore_N: Pop the first N callee-saved registers and ra, returns to ra

__riscv_save_N uses the alternate link register t0 so calling these does not have to destroy ra:

```
__riscv_save_1:
    addi sp, sp, -16
    sd s0, 0(sp)
    sd ra, 8(sp)
    jr t0
```

Zcmp extension

The Zcmp extension[3] defines `cm.push`, `cm.pop` etc

```
bar:
    cm.push   {ra, s0}, -16
    # ...
    cm.popret {ra, s0}, 16
```

(ARM: `push {lr, ...}` and `pop {pc, ...}`)

[3] https://github.com/riscv/riscv-code-size-reduction

Zcmp extension

```
# cm.push {ra, s0}, -16
sd s0, -8(sp)
sd ra, -16(sp)
addi sp, sp, -16
```

A small problem

Zcmp stack frame layout

Zcmp stack frame layout:

```
    # cm.push {ra, s0}, -16
    sd s0, -8(sp)
    sd ra, -16(sp)
    addi sp, sp, -16
```

Frame pointer convention and `__riscv_{save,restore}_N`:

```
answer:
    addi sp, sp, -16
    sd ra, 8(sp)
    sd s0, 0(sp)
```

Stack frame layout

- Zcmp is incompatible with `-fno-omit-frame-pointer` — must use longer sequence in this case
- `-msave-restore -fno-omit-frame-pointer` causes GCC (as of 12.2.0) to ignore `-msave-restore`, while LLVM inserts a `fp` update after `__riscv_save_N`

Conclusion

Conclusion

- There's lots of freedom in the base RISC-V calling convention about how the stack is structured
- Frame pointer convention essential for fast unwinding (`perf`, garbage collection, easy stack backtraces ...)
- Unfortunately most reliable unwinding is still DWARF...

References

- Calling convention: https://github.com/riscv-non-isa/riscv-elf-psabi-doc
- Zcmp: https://github.com/riscv/riscv-code-size-reduction

Thank you