RISC-V RV32 Calling Convention

hanlt2025@shanghaitech.edu.cn

Outline

Overview

From C to Registers

Preserving Control and State

Tools and Wrap-up

Appendix

Scope and Objectives

- Scope: standard RV32 integer calling convention first
- Focus: call instructions, register roles, stack frames, and compiler output
- Objective: understand the mechanisms that allow functions to call each other safely

Official references used

- RISC-V ABIs Specification (psABI)
- RISC-V Unprivileged ISA Manual

Roadmap

- C example and function-call requirements
- `jal`, `jalr`, `call`, and `ret`
- Register calling convention
- Example: nested calls
- Stack, stack frames, and saved values
- Exam example: recursive calling convention
- Godbolt and summary

C Example

```c
int g(int y) {
    return y + 1;
}

int f(int x, int y) {
    return x + g(y);
}
```

- `f` receives two arguments: `x` and `y`.
- Inside `f`, the call to `g(y)` happens before `f` can finish its own return value.
- Therefore, `f` is both a callee and a caller.

Calling-convention requirement

- The generated code must specify where **arguments**, **return values**, and **live registers** are placed when `f` calls `g`.

Requirements Imposed by the C Function Call

- The call instructions explain how control transfers from `f` to `g` and back.
- But they do not, by themselves, say where `x`, `y`, the return value, or the saved state should live.
- To compile C functions correctly, all code must follow the same register and stack rules.

Calling-convention components

- which registers carry **arguments**
- which registers carry **return values**
- which registers may be **clobbered** by a call
- which registers must be **preserved** across a call

JAL, JALR, `call`, and `ret`

Core instructions

- `jal rd, label`: `rd := pc + 4`, then jump to `label`
- `jalr rd, rs1, imm`: `rd := pc + 4`, then jump to `rs1 + imm`

```
jal   ra, callee
jalr x0, 0(ra)   # ret
```

Pseudo instructions

- `call foo` is assembler shorthand for a call sequence that links through `ra`
- `ret` is shorthand for `jalr x0, 0(ra)`

Instruction effect

- A call jumps to the callee and remembers where to return.
- A return jumps back through the address stored in `ra`.

Arguments and Return Values

- In standard RV32, `a0-a7` hold up to 8 integer or pointer arguments.
- `a0` and `a1` are also the integer return-value registers.
- In RV32, one register usually holds one 32-bit scalar.
- When argument registers run out, later arguments go on the stack.

Example mapping

```
a -> a0    e -> a4    i -> stack
b -> a1    f -> a5
c -> a2    g -> a6
d -> a3    h -> a7
```

psABI specification (psABI §2.1)

- The first stack-passed argument is at `0(sp)` on function entry.

Register Calling Convention: Quick Reference

Register	ABI Name	Role	Saved By
`x0`	`zero`	Hardwired zero	–
`x1`	`ra`	Return address	Caller
`x2`	`sp`	Stack pointer	Callee
`x8`	`s0/fp`	Saved register or frame pointer	Callee
`x9`	`s1`	Saved register	Callee
`x10-x17`	`a0-a7`	Integer arguments and return values	Caller
`x18-x27`	`s2-s11`	Saved registers	Callee
`x5-x7`, `x28-x31`	`t0-t6`	Temporaries	Caller

psABI specification (psABI §1.1)

- The ABI names `x1` as `ra`, `x2` as `sp`, `x8` as `s0/fp`, and `x10-x17` as `a0-a7`.
- “Registers `s0-s11` shall be preserved across procedure calls.”

Interpretation of Register Names

- `a` = argument / return-value registers
- `t` = temporary registers
- `s` = saved registers
- `ra` = return address
- `sp` = stack pointer
- `fp` = frame pointer, usually the same hardware register as `s0`

- This naming convention is part of the ABI, not part of the raw ISA.

Caller-saved vs Callee-saved

Caller-saved

- `ra`, `a0-a7`, `t0-t6`
- If the caller still needs one of these after a nested call, the caller must save it first.

Return-address remark

- `ra` is classified as caller-saved even though `ret` uses it.
- Once the callee makes another call, `ra` is overwritten.

Callee-saved

- `sp`, `s0-s11`
- If the callee changes one of these, it must restore it before returning.

Practical summary

- `a` and `t` registers are usually temporary.
- `s` registers are for values that must survive a nested call.

Incorrect and Correct Nested Call Handling

Broken

```
# int f(int x, int y) { return x + g(y); }
# a0=x, a1=y
f:
    mv   t0, a0
    mv   a0, a1
    call g
    add  a0, a0, t0
    ret
```

- `f` calls `g` but does not preserve `ra`.
- `t0` is also caller-saved, so `x` is not protected across the nested call.

Fixed

```
# int f(int x, int y) { return x +
g(y); }
# a0=x, a1=y
f:
    addi sp, sp, -16
    sw   ra, 12(sp)
    sw   s0, 8(sp)
    mv   s0, a0
    mv   a0, a1
    call g
    add  a0, a0, s0
    lw   s0, 8(sp)
    lw   ra, 12(sp)
    addi sp, sp, 16
    ret
```

- The corrected version preserves `ra` and uses `s0` to keep `x` live across the nested call.

Failure Mechanism of the Incorrect Version

Moment	`ra` now points to	What happens next
After `main` does `call f`	the instruction after the call in `main`	`pc` jumps to `f`
After `f` does `call g`	the instruction after that call inside `f`	`pc` jumps to `g`
When `g` executes `ret`	still inside `f`	control returns to `f`
When `f` later executes `ret`	still the same internal address	it returns into `f` instead of back to `main`

- The nested call overwrites `ra`.
- The local arithmetic may still be correct, but the control flow is no longer correct.

Locations for Preserved Values

Saved registers

- `s0-s11` are for values that must survive nested calls.
- If a callee modifies one of them, it must restore it before returning.

Stack frame

- `ra` and extra live values are often written to the stack.
- Local variables and overflow arguments also live there.

- In practice, preservation usually means either keeping the value in an `s` register or storing it in the stack frame.

Stack Basics on RV32

- The stack grows toward lower addresses.
- Additional arguments may be passed on the stack.
- Saved registers and local variables also live in the stack frame.
- Many function-call bugs are really stack-management bugs.

Simple picture

```
High addr
caller frame
-----------
stack arguments
-----------
saved ra / saved s regs
locals / spill slots
-----------
Low addr
```

Stack Alignment Requirement

- The standard RISC-V ABI requires `sp` to be 16-byte aligned on procedure entry.
- Keep that alignment across calls, not just at the very start of the program.
- Even if your function only uses `lw` and `sw`, library code may still depend on the ABI rule.

- Stack alignment is a correctness requirement, not a stylistic preference.

Frame Pointer and One Common RV32 Frame Layout

One common frame layout

```
High addr
stack args from caller   0(fp), 4(fp), ...
----------------------
saved ra                -4(fp)
saved old s0/fp         -8(fp)
local / spill slot    -12(fp), -16(fp), ...
----------------------
sp after prologue = fp - 16
Low addr
```

Typical RV32 prologue

```
addi sp, sp, -16
sw   ra, 12(sp)
sw   s0, 8(sp)
addi s0, sp, 16
```

psABI frame-pointer convention

- If a frame pointer is used, it must be `x8`, also named `s0/fp`.
- Many simple leaf functions omit `fp`, but compiler-generated code often uses it.

Leaf vs Non-leaf Functions

Leaf

```
# int add3(int x, int y, int z)
# a0=x, a1=y, a2=z
add3:
    add  a0, a0, a1
    add  a0, a0, a2
    ret
```

- No nested call, so `ra` is not overwritten after entry.

Non-leaf

```
# int f(int x, int y) { return x +
g(y); }
# a0=x, a1=y
f:
    addi sp, sp, -16
    sw   ra, 12(sp)
    sw   s0, 8(sp)
    mv   s0, a0
    mv   a0, a1
    call g
    add  a0, a0, s0
    lw   s0, 8(sp)
    lw   ra, 12(sp)
    addi sp, sp, 16
    ret
```

- Nested call: save `ra`, and restore any modified saved registers.

Preservation via Register or Stack Storage

```
# ABI-correct RV32 stack-based variant
f:
    addi sp, sp, -16
    sw   ra, 12(sp)
    sw   a0, 8(sp)
    mv   a0, a1
    call g
    lw   t0, 8(sp)
    add  a0, a0, t0
    lw   ra, 12(sp)
    addi sp, sp, 16
    ret
```

- This is still ABI-correct.
- It preserves `ra`, keeps `sp` aligned, and keeps `x` alive across the nested call.
- It is often slower than using `s0`, because it needs extra memory accesses.

Key idea

- The ABI cares about correctness of the contract, not whether you use a saved register or a stack slot.

Exam Example: Recursive Calling Convention

Source problem

- Adapted from CS110 2025 Midterm 1, Question 8.
- Recurrence: `f(n) = f(n-1) + f(n-2)`.
- Base case: when `n <= 2`, return `n`.

```c
int f(int n) {
    if (n <= 2) return n;
    return f(n - 1) + f(n - 2);
}
```

Problem skeleton

```
f:
    # A: Prologue
    ...
    # Termination condition: if (n <= 2)
return n;
    li   t0, 2
    ble  a0, t0, base_case
    # Compute f(n-1)
    addi a0, a0, -1
    call f
    # Call f(n-1)
    mv   s0, a0
    # Save return value to s0
    # B: Pass parameters for f(n-2)
    ...
    call f
    # Call f(n-2)
    add  a0, a0, s0
    # C: Epilogue (restoring saved
registers)
```

Exam Example: Recursive Calling Convention

```
    ...
    ret
base_case:
    # D: Set the correct return value
    ...
    # E: Epilogue
    ...
    ret
```

Values that must survive the second recursive call

- `ra`, because `f` is non-leaf.
- The original `n`, because it is needed again to form `n - 2`.
- `s0`, because it stores `f(n-1)` for use after the next call.

Exam Example: Minimal Correct Fill-in

A: Prologue

```
addi sp, sp, -12
sw   ra, 8(sp)
sw   a0, 4(sp)
sw   s0, 0(sp)
```

B: Pass parameters for `f(n-2)`

```
lw   a0, 4(sp)
addi a0, a0, -2
```

C: Epilogue

```
lw   s0, 0(sp)
lw   ra, 8(sp)
addi sp, sp, 12
```

D: Set the correct return value

```
# no extra instruction needed
```

E: Epilogue

```
lw   s0, 0(sp)
lw   ra, 8(sp)
addi sp, sp, 12
```

- This is the minimal ABI-correct answer under the exam's stated simplification.
- The exam explicitly says to ignore stack alignment, so it uses `12` bytes instead of the standard 16.

Observing Calling-Convention Code Generation in Godbolt

Procedure

- Open godbolt.org
- Write a small C function such as `int f(int x, int y) { return x + g(y); }`
- Choose a RISC-V compiler and inspect the generated assembly
- Compare `-O0` and `-O2`

Observation points

- how arguments are mapped to `a0`, `a1`, …
- whether the function is leaf or non-leaf
- whether `ra` and `s0` are saved in the prologue
- how `sp` changes and when the frame is released

- Godbolt makes it possible to inspect how the compiler realizes calling-convention rules in generated assembly.

Summary and References

- `jal`, `jalr`, `call`, and `ret` move control, but the ABI makes cooperation safe.
- In standard RV32, `a0-a7` carry integer arguments and `a0/a1` carry return values.
- `ra`, `a0-a7`, and `t0-t6` are caller-saved; `sp` and `s0-s11` are callee-saved.
- Non-leaf functions usually save `ra`, and stack alignment is mandatory.

Official references used in this lesson

- RISC-V ABIs Specification – mainly §1.1, §1.2, §2.1
- RISC-V Instruction Set Manual, Volume I – mainly §2.5
- Compiler Explorer (Godbolt) – for inspecting compiler-generated assembly

Appendix A: Meaning of “processor-specific” in psABI

- The official psABI introduction says: “This specification provides the processor-specific application binary interface document for RISC-V.” [1]
- Here, **processor-specific** means **specific to the RISC-V architecture family**, not to one vendor or one chip model.
- It is the RISC-V-specific ABI layer that sits on top of more generic binary-interface ideas such as ELF / gABI.

Mailing-list clarification

- Bruce Hoult explained in 2017 that the processor-specific ABI is about how functions from separately compiled object files and libraries call each other, not about the system-call ABI. [2]
- In this context, the psABI concerns **function-call interoperability** at the binary level.

Operational distinction

- ISA answers: “What does this instruction do?”
- psABI answers: “How do independently compiled RISC-V functions cooperate safely?”

Appendix B: 128-bit (= 16-byte) Stack Alignment

- The standard RISC-V psABI requires `sp` to be aligned to a 128-bit boundary on procedure entry. [1]
- Historically, a 2017 RISC-V SW Dev discussion says 16-byte alignment was chosen because of the possible presence of the `Q` extension; Andrew Waterman added that `RV32Q` was legal when that decision was made. [3]
- A later psABI-group discussion in 2022 gives a practical current reason: in ILP32, `long double` is 128 bits, so 16-byte stack alignment matches the alignment needs of that standard type. [4]

What the current spec shows

- In ILP32, `long double` has size 16 and alignment 16, and `max_align_t` is also aligned to 16. [1]
- `Tag_RISCV_stack_align` documents that the default stack alignment is 16 for RV32I / RV64I, but 4 for RV32E. [1]

- So the 16-byte rule is best understood as an ABI design choice for interoperability and type/layout consistency, not as a property forced by every RV32 instruction.

Appendix References (1/2)

[1] Official psABI specification

- K. Cheng and J. Clarke, eds., **RISC-V ABIs Specification**, Version 1.1 (pre-release), RISC-V International, Feb. 5, 2026. Available: official psABI site

[2] Mailing-list discussion: psABI vs syscall ABI

- B. Hoult, reply in “ABI for Ecall?”, **RISC-V SW Dev** mailing list, Jun. 29, 2017. Available: mailing-list thread link

[3] Mailing-list discussion: why 16-byte alignment

- B. Hoult, A. Bradbury, S. O’Rear, A. Waterman, et al., “Towards a ‘golden model’ of the RISC-V calling convention(s),” **RISC-V SW Dev** mailing list, Sep. 2017. Available: mailing-list thread link

Appendix References (2/2)

[4] Mailing-list discussion: EABI and long double motivation

- K. Cheng, “Seeking inputs for EABI design,” **RISC-V SW Dev** mailing list, Jan. 18, 2022. Available: mailing-list thread link

[5] Official ISA manual

- RISC-V International, **The RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture**. Available: official ISA PDF