RISC-V Calling Conventions

6.1810 Fall 2024

C code is compiled to machine instructions

How does the machine work at a lower level?

How does this translation work?

How to interact between C and asm?

```
Thread 1 hit Breakpoint 1, syscall () at kernel/syscall.
c:165
165         num = * (int *) 0;
```

```
    0x80001c24 <syscall+14> jal     ra,0x80000cf8 <myp
    0x80001c28 <syscall+18> mv      s1,a0
B+> 0x80001c2a <syscall+20> lw      s2,0(zero) # 0x0
    0x80001c2e <syscall+24> addiw   a4,s2,-1
    0x80001c32 <syscall+28> li      a5,21
    0x80001c34 <syscall+30> bltu    a5,a4,0x80001c82 <
```

RISC-V abstract machine

No C-like control flow, no concept of variables, types ...

Base ISA: Program counter, 32 general-purpose registers (x0 – x31)

RISC-V abstract machine

No C-like control flow, no concept of variables, types ...

Base ISA: Program counter, 32 general-purpose registers (x0 – x31)

reg	name	saver	description
x0	zero		hardwired zero
x1	ra	caller	return address
x2	sp	callee	stack pointer
x3	gp		global pointer
x4	tp		thread pointer
x5-7	t0-2	caller	temporary registers
x8	s0/fp	callee	saved register / frame pointer
x9	s1	callee	saved register
x10-11	a0-1	caller	function arguments / return values
x12-17	a2-7	caller	function arguments
x18-27	s2-11	callee	saved registers
x28-31	t3-6	caller	temporary registers
pc			program counter

Translating C to Assembly

Example: sum_to(n)

```
int sum_to(int n) {
  int acc = 0;
  for (int i = 0; i <= n; i++) {
    acc += i;
  }
  return acc;
}
```

What does this look like in assembly code?

```c
int sum_to(int n) {
    int acc = 0;

    for (int i = 0; i <= n; i++) {
        acc += i;
    }
    return acc;
}
```

```
# sum_to(n)
# expects argument in a0
# returns result in a0
sum_to:
  mv t0, a0            # t0 <- a0
  li a0, 0             # a0 <- 0
loop:
  add a0, a0, t0       # a0 <- a0 + t0
  addi t0, t0, -1      # t0 <- t0 - 1
  bnez t0, loop        # if t0 != 0: pc <-
loop
  ret
```

Limited abstractions in assembly

No local variables and scopes

Only a fixed set of hardware registers

- RISC-V has 32 registers
- Modern CPUs and GPUs have much more

No typed, positional arguments

Limited abstractions in assembly

No local variables and scopes

Only a fixed set of hardware registers

- RISC-V has 32 registers
- Modern CPUs and GPUs have much more

No typed, positional arguments

Assembly instructions directly translated to machine code

- Each instruction translated to 16 or 32 bit sequence

Calling and Returning Functions in Assembly

How would another function call `sum_to`?

Example main function

```
main:
  li a0, 10
  call sum_to
```

How does the call to `sum_to` work?

How would another function call `sum_to`?

Example main function

```
main:
 li a0, 10
 call sum_to
```

How does the call to sum_to work?

```
call {label} :=
   ra <- pc + 4          ; ra <- address of next instruction
   pc <- {label}         ; jump to {label}
```

Machine doesn't understand labels – translated to either `pc`-relative or absolute jumps

What are the semantics of return?

```
ret :=
  pc <- ra
```

What are the semantics of return?

```
ret :=
  pc <- ra
```

```
(  call {label} :=
       ra <- pc + 4
       pc <- {label}  )
```

Calling Convention & the Stack

Limited Registers Can Create Problems

Only 32 registers in RISC V

Function calls another function

- Called function can use and overwrite some register values
- When we return to the original function, it’s register values have been corrupted

Need a convention on who saves what registers?

Where do we save register values?

Example: what's the bug? (hint: it's on the right)

```
sum_to:
 mv t0, a0           # t0 <- a0
 li a0, 0            # a0 <- 0
loop:
 add a0, a0, t0      # a0 <- a0 + t0
 addi t0, t0, -1     # t0 <- t0 - 1
 bnez t0, loop       # if t0 != 0: pc <- loop
 ret
```

```
 # sum_then_double(n): expects argument in a0,
returns result in a0
 # expects argument in a0
 # returns result in a0
 sum_then_double:
   call sum_to
   li t0, 2             # t0 <- 2
   mul a0, a0, t0       # a0 <- a0 * t0
   ret

 main:
   li a0, 10
   call sum_then_double
```

What’s the issue with the program?

We get an infinite loop

Why?

```
sum_then_double:
  call sum_to
  li t0, 2              # t0 <- 2
  mul a0, a0, t0        # a0 <- a0 * t0
  ret

main:
  li a0, 10
  call sum_then_double
```

What's the issue with the program?

We get an infinite loop

Why?

- `sum_then_double` calls the function `sum_to`
- Sets the `ra` register to instruction immediately after `(li t0, 2)`
- Return in `sum_then_double` sets `pc` to `ra`, but that does not return it to main

```
sum_then_double:
  call sum_to
  li t0, 2            # t0 <- 2
  mul a0, a0, t0      # a0 <- a0 * t0
  ret

main:
  li a0, 10
  call sum_then_double
```

Fixing the bug

Save `ra` onto the stack in `sum_them_double`

- Allocate space on the stack
- Save `ra` onto the stack (before calling `sum_to`)
- At the end, load `ra` from the stack
- Deallocate stack space

Required coordination between caller and callee on who saves what

Need convention if including programs from different sources

```
sum_then_double:
    addi sp, sp, 16     # function prologue:
    sd ra, 0(sp)        # make space on stack,
                        # save registers
    call sum_to
    li t0, 2
    mul a0, a0, t0
    ld ra, 0(sp)        # function epilogue:
    addi sp, sp, -16    # restore registers,
                        # restore stack pointer
    ret
```

Calling Convention

Conventions surrounding this: "calling convention"

- How are arguments passed? `a0, a1, ..., a7,` rest on the stack
- How are values returned? `a0, a1`

Calling Convention

Conventions surrounding this: "calling convention"

- How are arguments passed? `a0, a1, ..., a7,` rest on the stack
- How are values returned? `a0, a1`

Who saves registers?

- Designated as *caller* or *callee* saved
- Q. Could `ra` be a *callee*-saved register?

Calling Convention

- Conventions surrounding this: "calling convention"
  - How are arguments passed? `a0, a1, ..., a7,` rest on the stack
  - How are values returned? `a0, a1`
- Who saves registers?
  - Designated as *caller* or *callee* saved
- Our assembly code should follow this convention
  - C code generated by GCC follows this convention
    - → This means that everyone's code can interop

Summary of Calling Convention

reg	name	saver	description
x0	zero		hardwired zero
x1	ra	caller	return address
x2	sp	callee	stack pointer
x3	gp		global pointer
x4	tp		thread pointer
x5–7	t0–2	caller	temporary registers
x8	s0/fp	callee	saved register / frame pointer
x9	s1	callee	saved register
x10–11	a0–1	caller	function arguments / return values
x12–17	a2–7	caller	function arguments
x18–27	s2–11	callee	saved registers
x28–31	t3–6	caller	temporary registers
pc			program counter

Stack Pointer & Frame Pointer

The Stack, Visually

```
Stack
                    .
                    .
     +->            .
     |    +-----------------+    |
     |    | return address  |    |
     |    |   previous fp -------+
     |    | saved registers |
     |    | local variables |
     |    |       ...       | <-+
     |    +-----------------+    |
     |    | return address  |    |
     +------- previous fp   |    |
          | saved registers |    |
          | local variables |    |
     +->  |       ...       |    |
     |    +-----------------+    |
     |    | return address  |    |
     |    |   previous fp -------+
     |    | saved registers |
     |    | local variables |
     |    |       ...       | <-+
     |    +-----------------+    |
     |    | return address  |    |
     +------- previous fp   |    |
          | saved registers |    |
          | local variables |    |
 $fp -->  |       ...       |    |
          +-----------------+    |
          | return address  |    |
          |   previous fp -------+
          | saved registers |
 $sp -->  | local variables |
          +-----------------+
```

The Stack, Visually

Composed of many stack frames, growing downward (hi → lo)

```
Stack
                    .
                    .
      +->           .
      |   +------------------+   |
      |   | return address   |   |
      |   |   previous fp -------+
      |   | saved registers  |
      |   | local variables  |
      |   |       ...        | <-+
      |   +------------------+   |
      |   | return address   |   |
      +------ previous fp    |   |
          | saved registers  |   |
          | local variables  |   |
      +-> |       ...        |   |
      |   +------------------+   |
      |   | return address   |   |
      |   |   previous fp -------+
      |   | saved registers  |
      |   | local variables  |
      |   |       ...        | <-+
      |   +------------------+   |
      |   | return address   |   |
      +------ previous fp    |   |
          | saved registers  |   |
          | local variables  |   |
  $fp --> |       ...        |   |
          +------------------+   |
          | return address   |   |
          |   previous fp -------+
          | saved registers  |
  $sp --> | local variables  |
          +------------------+
```

The Stack, Visually

Composed of many stack frames, growing downward (hi → lo)

- `sp` points to base of current stack
- `fp` points to end of previous stack frame

```
Stack
                    .
                    .
       +->          .
       |   +-----------------+   |
       |   | return address  |   |
       |   |   previous fp ------+
       |   | saved registers |
       |   | local variables |
       |   |       ...       | <-+
       |   +-----------------+   |
       |   | return address  |   |
       +------ previous fp   |   |
           | saved registers |   |
           | local variables |   |
       +-> |       ...       |   |
       |   +-----------------+   |
       |   | return address  |   |
       |   |   previous fp ------+
       |   | saved registers |
       |   | local variables |
       |   |       ...       | <-+
       |   +-----------------+   |
       |   | return address  |   |
       +------ previous fp   |   |
           | saved registers |   |
           | local variables |   |
   $fp --> |       ...       |   |
           +-----------------+   |
           | return address  |   |
           |   previous fp ------+
           | saved registers |
   $sp --> | local variables |
           +-----------------+
```

The Stack, Visually

Composed of many stack frames, growing downward (hi → lo)

- `sp` points to base of current stack
- `fp` points to end of previous stack frame

Why the frame pointer is useful

- The previous frame's `fp` is a fixed offset (-16) from the current frame's `fp`
- The return address (`ra`) lives at a fixed offset (-8) from the current frame's `fp`

```
Stack
                    .
                    .
      +->           .
      |     +-----------------+   |
      |     | return address  |   |
      |     |   previous fp ------+
      |     | saved registers |
      |     | local variables |
      |     |       ...       | <-+
      |     +-----------------+   |
      |     | return address  |   |
      +------ previous fp     |   |
            | saved registers |   |
            | local variables |   |
      +->   |       ...       |   |
      |     +-----------------+   |
      |     | return address  |   |
      |     |   previous fp ------+
      |     | saved registers |
      |     | local variables |
      |     |       ...       | <-+
      |     +-----------------+   |
      |     | return address  |   |
      +------ previous fp     |   |
            | saved registers |   |
            | local variables |   |
  $fp -->   |       ...       |   |
            +-----------------+   |
            | return address  |   |
            |   previous fp ------+
            | saved registers |
  $sp -->   | local variables |
            +-----------------+
```

The Stack, Visually

Composed of many stack frames, growing downward (hi → lo)

- `sp` points to base of current stack
- `fp` points to end of previous stack frame

Why the frame pointer is useful

- The previous frame's `fp` is a fixed offset (-16) from the current frame's `fp`
- The return address (`ra`) lives at a fixed offset (-8) from the current frame's `fp`

Can traverse previous stack frames by repeatedly finding the previous `fp`, and get useful information out of it (e.g. `ra`)

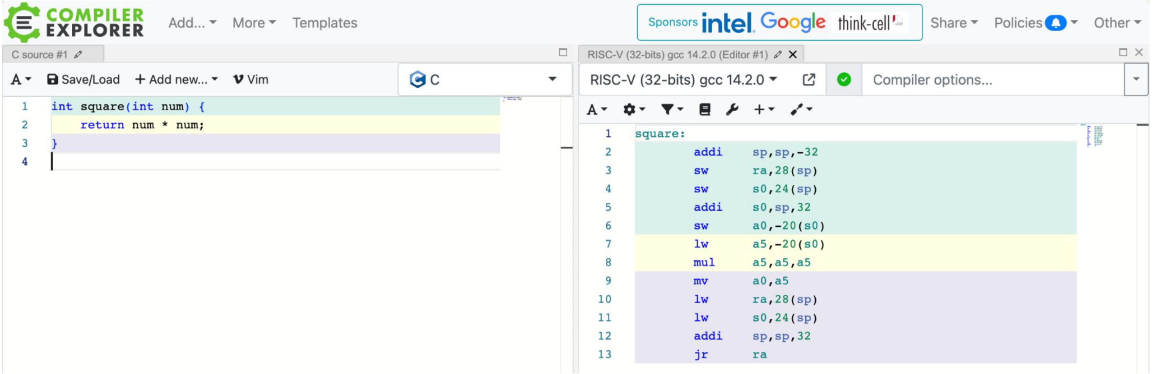
You'll do this in lab traps!

```
Stack
                   .
                   .
      +->          .
      |   +-----------------+   |
      |   | return address  |   |
      |   |   previous fp ------+
      |   | saved registers |
      |   | local variables |
      |   |       ...       | <-+
      |   +-----------------+   |
      |   | return address  |   |
      +------ previous fp   |   |
          | saved registers |   |
          | local variables |   |
      +-> |       ...       |   |
      |   +-----------------+   |
      |   | return address  |   |
      |   |   previous fp ------+
      |   | saved registers |
      |   | local variables |
      |   |       ...       | <-+
      |   +-----------------+   |
      |   | return address  |   |
      +------ previous fp   |   |
          | saved registers |   |
          | local variables |   |
  $fp --> |       ...       |   |
          +-----------------+   |
          | return address  |   |
          |   previous fp ------+
          | saved registers |
  $sp --> | local variables |
          +-----------------+
```

Resources

Resources

[Godbolt]



GDB

```
0x0000000000001000 in ?? ()
(gdb) b syscall
Breakpoint 1 at 0x80001c16: file kernel/syscall.c, line
160.
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, syscall () at kernel/syscall.
c:160
warning: Source file is more recent than executable.
160     {
```

[6.191 RISC-V ISA Reference Card]