```
6.S081/6.828 2019 Lecture 11: RISC-V calling convention, stack frames, and gdb

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
  Why this matters: sometimes need to write code not expressible in C
    And you need this for the syscall lab!

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

Running 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?

  # 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
  No typed, positional arguments
  No local variables
  Only registers

Machine doesn't even see assembly code
  Sees binary encoding of machine instructions
    Each instruction: 16 bits or 32 bits
  E.g. `mv t0, a0` is encoded as 0x82aa
  Not quite 1-to-1 encoding from asm, but close
```

```
How would another function call sum_to?

  main:
    li a0, 10          # a0 <- 10
    call sum_to

What are the semantics of call?

  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

Let's try it out: demo1.S
  (gdb) file user/_demo1
  (gdb) break main
  (gdb) continue
  Why does it stop before running demo1?
  (gdb) layout split
  (gdb) stepi
  (gdb) info registers
  (gdb) p $a0
  (gdb) advance 18
  (gdb) si
  (gdb) p $a0

What if we wanted a function calling another function?

  # sum_then_double(n)
  # 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

Let's try it out: demo2.S
  We get stuck in an infinite loop
  Why: overwrote return address (ra)

How to fix: save ra somewhere
  In another register? Won't work, just defers problem.
  Solution: save on stack

  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
```

```
Let's try it out: demo3.S
  (gdb) ...
  (gdb) nexti

So far, our functions coordinated with each other
  This worked because we were writing all the code involved
  Could have written it any other way
    E.g. passing arguments in t2, getting return value in t3

Conventions surrounding this: "calling convention"
  How are arguments passed?
    a0, a1, ..., a7, rest on stack
  How are values returned?
    a0, a1
  Who saves registers?
    Designated as caller or callee saved
    Could ra be a callee-saved register?
  Our assembly code should follow this convention
  C code generated by GCC follows this convention
  This means that everyone's code can interop, incl C/asm interop
  Read: demo4.c / demo4.asm
    Can see function prologue, body, epilogue
    Why doesn't it save ra? Leaf function, not needed
    What is going on with s0/fp?
      We compiled with -fno-omit-frame-pointer

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 |
          +-----------------+

Demo program: demo5.c
  (gdb) break g
  (gdb) si
```

```
  (gdb) si
  (gdb) si
  (gdb) si
  (gdb) p $sp
  (gdb) p $fp
  (gdb) x/g $fp-16
  (gdb) x/g 0x0000000000002fd0-16

Stack diagram:

          0x2fe0 |
          0x2fd8 | <garbage ra>       \
          0x2fd0 | <garbage fp>       / stack frame for main
          0x2fc8 | ra into main       \
  $fp --> 0x2fc0 | 0x0000000000002fe0 / stack frame for f
          0x2fb8 | ra into f          \
  $sp --> 0x2fb0 | 0x0000000000002fd0 / stack frame for g

GDB can automate this reasoning for us
  Plus, it can use debug info to reason about leaf functions, etc.
  (gdb) backtrace
  (gdb) info frame
  (gdb) frame 1
  (gdb) info frame
  (gdb) frame 2
  (gdb) info frame

Calling C from asm / calling asm from C
  Follow calling convention and everything will work out
  Write function prototype so C knows how to call assembly
  Demo: demo6.c / demo6_asm.S
    Why do we use s0/s1 instead of e.g. t0/t1?
    (gdb) b sum_squares_to
    (gdb) si ...
    (gdb) x/4g $sp
    (gdb) si ...

Inline assembly

Structs
  C struct layout rules
    Why: misaligned load/store can be slow or unsupported (platform-dependent)
  __attribute__((packed))
  How to access and manipulate C structs from assembly?
    Generally passed by reference
    Need to know struct layout
    Demo: demo7.c / demo7_asm.S

Debugging
  examine: inspect memory contents
    x/nfu addr
      n: count
      f: format
      u: unit size
  step/next/finish
    step: next line of C code
    next: next line of C code, skipping over function calls
    finish: continue executing until end of current function call
  stepi/nexti
    stepi: next assembly instruction
    nexti: next assembly instruction, skipping over function calls
  layout next
    steps through layouts
  conditional breakpoints
    break, only when a condition holds (e.g. variable has a certain value)
```

```
    watchpoints
      break when a memory location changes value
    GDB is a very powerful tool
      Read the manual for more!
      But you probably don't need all the fancy features for this class

  References
    RISC-V ISA specification: https://riscv.org/specifications/
      Contains detailed information
    RISC-V ISA Reference: https://rv8.io/isa
      Overview of instructions
    RISC-V assembly language reference: https://rv8.io/asm
      Overview of directives, pseudo-instructions, and more
```