Computer Organization and Architecture

RISC-V 64 Assembly Language

Utah Tech University

Spring 2025

Calling Functions

To implement functions/procedures, we must be able to:

- Jump to the code that implements the function
- Return to the jump site when it is finished
- Pass arguments to the function
- Get return values back from the function

Call and Return

- Here is code that
  - calls a function with two arguments (a0 and a1)
  - then uses the return value (a0) as the argument to the exit system call
- `jal` is the *jump and link* instruction. It
  - jumps to address 0x1088 (loads that address into the PC)
  - copies the address 0x1080 (the instruction following `jal`, where the function should return to) to the *return address*, a.k.a. `x1` or `ra`.
- The `ret` instruction
  - copies the value from ra back into the PC, effectively
  - jumping back to the the instruction after the `jal`
- The actual jump target address is computed by skipping forward 3 instructions or 12 bytes. The immediate field of the `jal` instruction would be 12 in this case (since it is added to the PC when the jump is executed), but the assembler and disassembler show this as the target address.

```
1074:    00500513        li   a0,5
1078:    00700593        li   a1,7
107c:    00c000ef        jal  ra,0x1088
1080:    05d00893        li   a7,93
1084:    00000073        ecall
1088:    00b50533        add  a0,a0,a1
108c:    00008067        ret
```

Call and Return

Here is the same code run through our simulator:

```
4212: 00500513    li      a0, 5                 a0 <- 5
4216: 00700593    li      a1, 7                 a1 <- 7
4220: 00c000ef    jal     12                    ra <- 4224, pc <- 4232
4232: 00b50533    add     a0, a0, a1            a0 <- 12
4236: 00008067    ret                           pc <- 4224
4224: 05d00893    li      a7, 93                a7 <- 93
4228: 00000073    ecall                         exit(12)
```

This traces the instructions in the order they are executed, while the disassembly shows them in the order the instructions are laid out in memory.

Call and Return

- Here is the original source code
  - `_start` is the entry point for every program
  - we use a label for the function `add2` and let the assembler compute the address
  - we use another label for the system call number
- Note the differences between the call and the return
  - `jal` has the (relative) address hard-coded in the instruction
    - it always branches to the **same** place
    - it encodes a relative address since the branch source and target are fixed
  - the return address is compute on-the-fly by `jal` and stored in a register for `ret` to use
    - the function could be called from many different places in the program
    - `jal` computes the full address 0x1080 and stores it in ra
    - it needs the ability to branch back to a **different** place each time
- What happens if `add2` needs to call a function?

```
_start:     li      a0, 5
            li      a1, 7
            jal     add2
            li      a7, sys_exit
            ecall
add2:       add     a0, a0, a1
            ret
```

Parameter Passing

- How do functions pass arguments?

  1. On the stack
  2. In registers
  3. In registers and on the stack

- *Calling conventions* are the rules functions follow to interoperate with each other. A function call requires coordination between the *caller* (the code initiating the function call) and the *callee* (the code being called).

- Each function allocates a chunk of stack space called a *stack frame* where it can store private data.
  Compilers/programmers have quite a bit of freedom in how they manage the stack frame, but it also has some structure that everyone must honor.

Parameters

- Arguments go in the registers `a0`–`a7` in order
- The return value is put in `a0`
- Note: nothing magic happens when a `jal` instruction is issued: it is up to the caller to put the arguments in the right place and then the callee trusts that they are there. Same for return value.
- Complications:
  - What if there are more than 8 arguments?
    - The remaining arguments go on the stack
  - What if an argument does not fit in a register (a `struct`)
    - The too-big argument goes on the stack

Registers

- `pc`: The program counter always contains the address of the instruction currently running. This is not one of the 32 registers you can use in normal instructions—only special instructions (branches, jumps, etc.) access it and only in special ways.
- `zero`: The zero register is a special case. Its value is always zero when read, and values written to it are discarded.
- `ra`: This return address register is where return addresses are stored by default by the `jal` instruction and we normally do not use it for anything else.
- `sp`: The stack pointer register is normally reserved for holding the address of the current bottom of the stack.
- `a0`–`a7`: The argument registers are used for passing parameters to functions, and then functions may use them freely as scratch registers. If the caller cares about their contents, the **caller** must save them before making a function call.
- `t0`–`t6`: The temporary registers are additional scratch registers that functions can use freely. If the caller cares about their contents when making a function call, it must save them before making the call.
- `s0`–`s11`: Saved registers—if the callee uses them, then the callee must restore the original values before returning. Standard practice is to store the old values on the stack frame at the beginning of the function, use them, then load the values back from the stack frame to restore them before returning.

Stack frames

- Each instance of a function allocates a *stack frame*, which is just a chunk of stack space owned by that instance. Stack frames typically hold:
  - The return address (from the link register)
  - Parameters that do not fit in the registers
  - Local variables
  - Copies of callee-saved registers
- Imagine a recursive function that calls itself many times. Each stack frame is tied to a single call, so there can be many instances of the function outstanding but their local storage will not overlap or be confused.

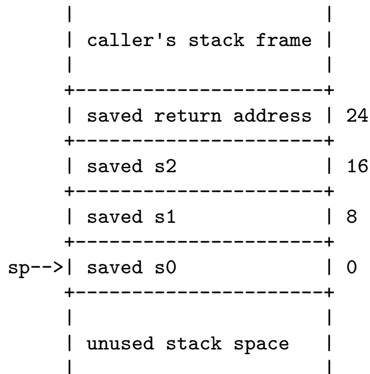
Example stack frame

Consider a typical stack frame and the code that sets it up:

```
myfunc:
        # function prelude:
        addi    sp, sp, -32
        sd      ra, 24(sp)
        sd      s2, 16(sp)
        sd      s1, 8(sp)
        sd      s0, 0(sp)

        # main function code goes here
        # can use s0, s1, and s2
        # and can make function calls

        # function postlude:
        ld      ra, 24(sp)
        ld      s2, 16(sp)
        ld      s1, 8(sp)
        ld      s0, 0(sp)
        addi    sp, sp, 32
        ret
```



The Goal

Our goal in learning assembly language is to better understand what actually happens when we write code in a high-level language. To that end we will start with a function in C or C-like pseudo-code and transform it into simplified code with these qualities:

- No indented block structure (only labels and goto statements). This means all loops and complicated if-else sequences must be transformed into simpler equivalents
- The only if-statements are comparisons of two numbers where the action is a goto statement
- Expressions are all simplified so that each line can be translated directly into one or two lines of assembly

Before you write a single line of actual assembly language, you should know the following:

- Every variable (including the intermediate results normally hidden in complex expressions) and everywhere it is used. This includes knowing if it is a value that needs to be in a saved register or if a temporary register will suffice
- How many registers are needed and which ones they are. This includes knowing how many saved registers need to be written to the stack in the function prelude and restored in the postlude.

The Process

- Start with a function in C or C-like pseudo-code
- Transform complex control flow into simple if-statements with goto statements, removing all indented block structure
- Transform other complex statements and expressions into simpler versions where every intermediate value has an explicit name
- Plan which variables will occupy which registers
- Convert the function line-by-line into real assembly language

Control flow

- High-level languages have various ways to control what happens next:
  - Conditionals: if, else if, else, etc.
  - Loops: while, do while, for, for range, etc.
  - Switch: switch and case
- In assembly language our basic tools is:
  - Compare two numbers (two registers or a register and an immediate constant)
  - Based on the result (equal, not equal, less than, less than or equal, greater than, greater than or equal), either branch (jump) or keep going
- We will transform high-level constructs into:
  - Labels that identify a spot in the code
  - `if` that compares two values followed by `goto`
  - No complex comparisons, no else, nothing else inside the `if` block

Simple if statements

- The problem: the `if` block has anything other than a `goto` inside it

```
if a > b:
    print("a is bigger")
print("back together")
```

- The solution: invert the test
  - Change “if condition do xyz” to “if !condition skip xyz”

```
    if a <= b:
        goto 1f
    print("a is bigger")
1:  print("back together")
```

If with else

- The problem: the `if` is followed by an `else`

```
if a > b:
    print("a is bigger")
else:
    print("a is not bigger")
print("back together")
```

- The solution: add a `goto` to skip over the `else` part

```
    if a <= b:
        goto 1f
    print("a is bigger")
    goto 2f
1:  print("a is not bigger")
2:  print("back together")
```

- Note: labels are not instructions and do not impact the control flow. It will fall through from the code at label `1` to the code at label `2`

Chains of else if

- The problem: `else if` chains
  - Pattern works for any number

```
if a > b:
    print("a is bigger")
elif a == b:
    print("a and b equal")
else:
    print("a is smaller")
print("back together")
```

- The solution: invert each test, skip to join point after each block

```
    if a <= b:
        goto 1f
    print("a is bigger")
    goto 3f
1:  if a != b:
        goto 2f
    print("a and b equal")
    goto 3f
2:  print("a is smaller")
3:  print("back together")
```

do while

- Not the most common type of loop, but the easiest to work with

```
do {
    printf("inside the loop\n");
    a++;
} while (a < b);
printf("finished\n");
```

- The test condition does *not* need to be inverted

```
1:  printf("inside the loop\n");
    a++;
    if (a < b)
        goto 1b;
    printf("finished\n");
```

while

- `do while` always runs at least once, but `while` can run zero or more times

```
while a < b:
    print("inside the loop")
    a += 1
print("finished")
```

- Like an `if` but with a branch at the bottom to start over

```
1:  if a >= b:
        goto 2f
    print("inside the loop")
    a += 1
    goto 1b
2:  print("finished")
```

C-style for loop

- C-style `for` loops transform into `while` loops

```
for (int i = 0; i < size; i++) {
    printf("element %d is %d\n", i, array[i]);
}
printf("finished\n");
```

- Carefully note where the update (`i++`) part fits in

```
    int i = 0;
1:  if (i >= size)
        goto 2f;
    printf("element %d is %d\n", i, array[i]);
    i++;
    goto 1b;
2:  printf("finished\n");
```

break

- `break` terminates a loop immediately

```
for (int i = 0; i < size; i++) {
    if (array[i] < 0)
        break;
    printf("element %d is %d\n", i, array[i]);
}
printf("finished\n");
```

- `break` is a form of `goto` so it can go inside an `if`

```
    int i = 0;
1:  if (i >= size)
        goto 2f;
    int temp = array[i];
    if (temp < 0)
        goto 2f
    printf("element %d is %d\n", i, array[i]);
    i++;
    goto 1b;
2:  printf("finished\n");
```

continue

- `continue` skips to the next iteration of the loop

```
for (int i = 0; i < size; i++) {
    if (array[i] < 0)
        continue;
    printf("element %d is %d\n", i, array[i]);
}
printf("finished\n");
```

- `continue` still performs the update and the test

```
    int i = 0;
1:  if (i >= size)
        goto 3f;
    int temp = array[i];
    if (temp < 0)
        goto 2f
    printf("element %d is %d\n", i, array[i]);
2:  i++;
    goto 1b;
3:  printf("finished\n");
```

Python for loops

- What about Python `for` loops?

```
for i in range(10):
    print(i)
print("finished")
```

- Typical `for` with `range` is similar to C

```
    i = 0
1:  if i >= 10:
        goto 2f
    print(i)
    i += 1
    goto 1b
2:  print("finished")
```

- `break` and `continue` work the same way as in C

Python for loops

- What about iterating over a collection?

```
for elt in lst:
    print(elt)
print("finished")
```

- Like the version using `range`, but you must find the length of the list and look up the element each iteration

```
    i = 0
    size = len(lst)
1:  if i >= size:
        goto 2f
    elt = lst[i]
    print(elt)
    i += 1
    goto 1b
2:  print("finished")
```

64-bit integers

RISC-V is a *load/store* architecture, meaning that the interface with memory is simple and limited:

- You can load a value from memory into a register

```
ld  dest, immediate offset(register with memory address)
ld  t0, 8(a1)
```

- You can store a value from a register into memory

```
sd  src, immediate offset(register with memory address)
sd  t1, 16(t2)
```

- If you need a more complex address calculation, you must compute it first and then issue the load or store instruction.

Examples

- Load a global variable into `t3` given a pointer in `a2`

```
ld  t3, (a2)
```

- Load a global variable with label `size`

```
la  t0, size
ld  t3, (t0)
```

- Load a value from an array into `t1`. The array pointer is in `a4` and the index is in `a7`. We use `t2` as a temporary register to compute the effective address:

```
slli    t2, a7, 3
add     t2, a0, t2
ld      t1, (t2)
```

  - `a0` specifies the base address
  - `a7` is an index, but each element is 8 bytes in size
  - `slli` shifts `a7` left 3 times, effectively multiplying `a7` by 8 and storing the result in `t2`
  - The scaled index and array base are added together to compute the *effective address* of the array element

- `slli` is a convenient way to scale index values by a power of 2

32-bit and 16-bit integers

You can also load and store 32-bit and 16-bit values:

- The registers are the same, but you can load a smaller value and either sign extend it to 64 bits (using `lw` or `lh`) or fill in the remaining bits with zeros (using `lwu` or `lhu`):

```
slli    t2, a7, 2       # 16 bit: slli  t2, a7, 1
add     t2, a0, t2      # 16 bit: same
lw      t1, (t2)        # 16 bit: lh    t1, (t2)
```

  - The address is still 64 bits
  - A 32/16-bit value is loaded into the lower bits of `t1` and the remaining bits are copies of the sign bit
  - Since each element in the array is only 4/2 bytes in size, we shift left twice/once (multiply by 4/2)

Strings

C strings are stored as an array of 1-byte characters with a zero byte marking the end

- To load a single byte

```
lb  t3, (a5)
```

  - Addresses are still 64 bits
  - The remaining 56 bits of the target register are filled in using sign extension (`lb`) or with zeros (`lbu`)

- The same process is used for array lookups

```
add     t2, a0, a7
lb      t1, (t2)
```

  - Since each element is 1 byte, there is usually no need to shift