Computer Organization and Architecture
RISC-V 32 Assembly Language

Dixie State University—Computing and Design

Spring 2022
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.
Here is the same code run through our simulator:

4212: 00500513_ li a0, 5  
4216: 00700593_ li a1, 7  
4220: 00c000ef_ jal 12  
4232: 00b50533_ add a0, a0, a1  
4236: 00008067_ ret  
4224: 05d00893_ li a7, 93  
4228: 00000073_ ecall  

a0 <- 5  
a1 <- 7  
ra <- 4224, pc <- 4232  
a0 <- 12  
pc <- 4224  
a7 <- 93  
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.
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?
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

- **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.
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:

```assembly
myfunc:

    # function prelude:
    addi sp, sp, -16
    sw ra, 12(sp)
    sw s2, 8(sp)
    sw s1, 4(sp)
    sw s0, 0(sp)

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

    # function postlude:
    lw ra, 12(sp)
    lw s2, 8(sp)
    lw s1, 4(sp)
    lw s0, 0(sp)
    addi sp, sp, 16
    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
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 if
print("a is bigger")
1:
print("back together")
The problem: the if is followed by an else

```python
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

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

The problem: else if chains
   Patterns 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

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

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

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

- Usually best to transform it into a do 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")
```

- Add a branch to the test at the top to allow zero runs
- Note: if the loop is guaranteed to run at least once, skip the goto at the top (e.g., a for loop with a fixed limit)
- Only one branch per iteration after that

```
goto 2f
1:
    print("inside the loop")
    a += 1
2:
    if a < b:
        goto 1b
    print("finished")
```
C-style for loops transform into while loops

```c
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

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

- **break terminates a loop immediately**
  
  ```c
  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**
  
  ```c
  int i = 0;
  goto 2f;
  1:
    int temp = array[i];
    if (temp < 0)
      goto 3f
    printf("element %d is %d\n", i, array[i]);
    i++;
  2:
    if (i < size)
      goto 1b;
  3:
    printf("finished\n");
  ```
**continue**

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

```c
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

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

- What about Python for loops?

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

- Typical for with range is similar to C

```c
i = 0
goto 2f

1:
    print(i)
i += 1

2:
    if i < 10:
        goto 1b
    print("finished")
```

- break and continue work the same way as in C
Python for loops

- What about iterating over a collection?

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

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

  ```python
  i = 0
  size = len(lst)
  goto 2f
  1:
      elt = lst[i]
      print(elt)
      i += 1
  2:
      if i < size:
          goto 1b
      print("finished")
  ```
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
  
  ```
  lw  dest, immediate offset(register with memory address)
  lw  t0, 8(a1)
  ```

- You can store a value from a register into memory
  
  ```
  sw  src, immediate offset(register with memory address)
  sw  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

  ```
  lw  t3, (a2)
  ```

- 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, 2
  add   t2, t2, a4
  lw    t1, (t2)
  ```

  - a4 specifies the base address
  - a7 is an index, but each element is 4 bytes in size
  - slli shifts a7 left 2 times, effectively multiplying a7 by 4 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
16-bit integers

You can also load and store 16-bit values:

- The registers are the same, but you can load a 16-bit value and either sign extend it to 32 bits (using `lh`) or fill in the remaining 16 bits with zeros (using `lhu`):

  ```
  slli t2, a7, 1
  add t2, t2, a4
  lh t1, (t2)
  ```

- The address is still 32 bits
- A 16-bit value is loaded into half of `t1` and the remaining 16 bits are copies of the sign bit
- Since each element in the array is only 2 bytes in size, we shift left once (multiply by 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
  
  \[ \text{lb } t3, (a5) \]
  
  - Addresses are still 32 bits
  - The remaining 24 bits of the target register are filled in using sign extension (lb) or with zeros (lbu)

- The same process is used for array lookups

\[
\begin{align*}
\text{add} & \quad t2, t2, a4 \\
\text{lb} & \quad t1, (t2)
\end{align*}
\]

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