CS 61C Spring 2021 — RISC-V Intro & Control Flow

Discussion 4: February 8, 2021

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 After calling a function and having that function return, the `t` registers may have been changed during the execution of the function, while `a` registers cannot.

False. `a0` and `a1` registers are often used to store the return value from a function, so the function can set their values to the its return values before returning.

1.2 Let `a0` point to the start of an array `x`. `lw s0, 4(a0)` will always load `x[1]` into `s0`.

False. This only holds for data types that are four bytes wide, like `int` or `float`. For data-types like `char` that are only one byte wide, `4(a0)` is too large of an offset to return the element at index 1, and will instead return a `char` further down the array (or some other data beyond the array, depending on the array length).

1.3 Assuming no compiler or operating system protections, it is possible to have the code jump to data stored at `0(a0)` (offset 0 from the value in register `a0`) and execute instructions from there.

True. If your compiler/OS allows it (some do not, for security reasons), it is possible for your code to jump to and execute instructions passed into the program via an array. Conversely, it's also possible for your code to treat itself as normal data (search up self-modifying code if you want to see more details).

1.4 Assuming integers are 4 bytes, adding the ASCII character `'d'` to the address of an integer array would get you the element at index 25 of that array (assuming the array is large enough).

True. There is no fundamental difference between integers, strings, and memory addresses in RISC-V (they're all bags of bits), so it's possible to manipulate data in this way. (We don't recommend it, though).

1.5 `jalr` is a shorthand expression for a `jal` that jumps to the specified label and does not store a return address anywhere.

False. `j label` is a pseudo-instruction for `jal x0, label`. `jalr` is used to return to the memory address specified in the second argument. Keep in mind that `jal` jumps to a label (which is translated into an immediate by the assembler), whereas `jalr` jumps to an address stored in a register, which is set at runtime.

1.6 Calling `j label` does the exact same thing as calling `jal label`.

False. As from the previous problem, `j label` is short for `jal x0, label` — since it's writing the return address to `x0`, it's effectively discarding it since we have no need to jump back to the original PC. `jal label` is short for `jal ra, label`.

2 Basic Instructions

For your reference, here are some of the basic instructions for arithmetic operations and dealing with memory (Note: ARG1 is argument register 1, ARG2 is argument register 2, and DR is destination register):

[inst]	[destination register] [argument register 1] [argument register 2]
add	Adds the two argument registers and stores in destination register
xor	Exclusive or's the two argument registers and stores in destination register
mul	Multiplies the two argument registers and stores in destination register
sll	Logical left shifts ARG1 by ARG2 and stores in DR
srl	Logical right shifts ARG1 by ARG2 and stores in DR
sra	Arithmetic right shifts ARG1 by ARG2 and stores in DR
slt/u	If ARG1 < ARG2, stores 1 in DR, otherwise stores 0, u does unsigned comparison

[inst]	[register] [offset]([register containing base address])
sw	Stores the contents of the register to the address+offset in memory
lw	Takes the contents of address+offset in memory and stores in the register

[inst]	[argument register 1] [argument register 2] [label]
beq	If ARG1 == ARG2, moves to label
bne	If ARG1 != ARG2, moves to label

[inst]	[destination register] [label]
jal	Stores the next instruction's address into DR and moves to label

You may also see that there is an "i" at the end of certain instructions, such as addi, slli, etc. This means that ARG2 becomes an "immediate" or an integer instead of using a register. There are also immediates in some other instructions such as **sw** and **lw**. Note that the size (maximum number of bits) of an immediate in any given instruction depends on what type of instruction it is (more on this soon!).

2.1 Assume we have an array in memory that contains `int *arr = {1,2,3,4,5,6,0}`.

Let register `s0` hold the address of the element at index 0 in arr. You may assume integers are four bytes and our values are word-aligned. What do the snippets of RISC-V code do? Assume that all the instructions are run one after the other in the same context.

```
a) lw   t0, 12(s0)        -->       Sets t0 equal to arr[3]

b) sw   t0, 16(s0)        -->       Stores t0 into arr[4]

c) slli t1, t0, 2
   add  t2, s0, t1
   lw   t3, 0(t2)         -->       Increments arr[t0] by 1
   addi t3, t3, 1
   sw   t3, 0(t2)

d) lw   t0, 0(s0)
   xori t0, t0, 0xFFF     -->       Sets t0 to -1 * arr[0]
   addi t0, t0, 1
```

3 C to RISC-V

3.1 Translate between the C and RISC-V verbatim.

C	RISC-V

`// s0 -> a, s1 -> b`
`// s2 -> c, s3 -> z`
`int a = 4, b = 5, c = 6, z;`
`z = a + b + c + 10;`
	`addi s0, x0, 4`
	`addi s1, x0, 5`
	`addi s2, x0, 6`
	`add  s3, s0, s1`
	`add  s3, s3, s2`
	`addi s3, s3, 10`

`// s0 -> int * p = intArr;`
`// s1 -> a;`
`*p = 0;`
`int a = 2;`
`p[1] = p[a] = a;`
	`sw   x0, 0(s0)`
	`addi s1, x0, 2`
	`sw   s1, 4(s0)`
	`slli t0, s1, 2`
	`add  t0, t0, s0`
	`sw   s1, 0(t0)`

`// s0 -> a, s1 -> b`
`int a = 5, b = 10;`
`if(a + a == b) {`
`    a = 0;`
`} else {`
`    b = a - 1;`
`}`
	`    addi s0, x0, 5`
	`    addi s1, x0, 10`
	`    add  t0, s0, s0`
	`    bne  t0, s1, else`
	`    xor  s0, x0, x0`
	`    jal  x0,  exit`
	`else:`
	`    addi s1, s0, -1`
	`exit:`

`// computes s1 = 2^30`
`// assume int s1, s0; was declared above`
`s1 = 1;`
`for(s0 = 0; s0 != 30; s0++) {`
`    s1 *= 2;`
`}`
	`    addi s0, x0, 0`
	`    addi s1, x0, 1`
	`    addi t0, x0, 30`
	`loop:`
	`    beq  s0, t0, exit`
	`    add  s1, s1, s1`
	`    addi s0, s0, 1`
	`    jal  x0, loop`
	`exit:`

`// s0 -> n, s1 -> sum`
`// assume n > 0 to start`
`for(int sum = 0; n > 0; n--) {`
`  sum += n;`
`}`
	`    addi s1, x0, 0`
	`loop:`
	`    beq  s0, x0, exit`
	`    add  s1, s1, s0`
	`    add  s0, s0, -1`
	`    jal  x0, loop`
	`exit:`

4 RISC-V with Arrays and Lists

Comment what each code block does. Each block runs in isolation. Assume that there is an array, `int arr[6] = {3, 1, 4, 1, 5, 9}`, which starts at memory address `0xBFFFFF00`, and a linked list struct (as defined below), `struct ll* lst`, whose first element is located at address `0xABCD0000`. Let `s0` contain `arr`'s address `0xBFFFFF00`, and let `s1` contain `lst`'s address `0xABCD0000`. You may assume integers and pointers are 4 bytes and that structs are tightly packed. Assume that `lst`'s last node's `next` is a NULL pointer to memory address `0x00000000`.

```
struct ll {
    int val;
    struct ll* next;
}
```

4.1
```
lw  t0, 0(s0)
lw  t1, 8(s0)
add t2, t0, t1
sw  t2, 4(s0)
```

Sets `arr[1]` to `arr[0]` + `arr[2]`.

4.2
```
loop: beq  s1, x0, end
      lw   t0, 0(s1)
      addi t0, t0, 1
      sw   t0, 0(s1)
      lw   s1, 4(s1)
      jal  x0, loop
 end:
```

Increments all values in the linked list by 1.

4.3
```
       add  t0, x0, x0
loop:  slti t1, t0, 6
       beq  t1, x0, end
       slli t2, t0, 2
       add  t3, s0, t2
       lw   t4, 0(t3)
       sub  t4, x0, t4
```

```
        sw   t4, 0(t3)
        addi t0, t0, 1
        jal  x0, loop
end:
```

Negates all elements in `arr`.

5 RISC-V Calling Conventions

5.1 How do we pass arguments into functions?

Use the 8 arguments registers `a0` - `a7`.

5.2 How are values returned by functions?

Use `a0` and `a1` as the return value registers.

5.3 What is `sp` and how should it be used in the context of RISC-V functions?

`sp` stands for stack pointer, and it represents the boundary between stored data and free space on the stack. Because the stack grows downward, we subtract from `sp` to create more space (moving the stack pointer down), and add to `sp` to free space (moving the stack pointer back up). The stack is mainly used to save (and later restore) the value of registers that may be overwritten.

5.4 Which values need to saved by the caller, before jumping to a function using `jal`?

Registers `a0` - `a7`, `t0` - `t6`, and `ra`.

5.5 Which values need to be restored by the callee, before returning from a function?

Registers `sp`, `gp` (global pointer), `tp` (thread pointer), and `s0` - `s11`. Note that we don't use `gp` and `tp` very often in this course.

5.6 In a bug-free program, which registers are guaranteed to be the same after a function call? Which registers aren't guaranteed to be the same?

Registers `a0` - `a7`, `t0` - `t6`, and `ra` are not guaranteed to be the same after a function call (which is why they must be saved by the caller). Registers `sp`, `gp`, `tp`, and `s0` - `s11` are guaranteed to be the same after a function call (which is why the callee must restore them before returning).

6 Writing RISC-V Functions

6.1 Write a function `sumSquare` in RISC-V that, when given an integer `n`, returns the summation below. If `n` is not positive, then the function returns 0.

$$n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$$

For this problem, you are given a RISC-V function called `square` that takes in a single integer and returns its square.

First, let's implement the meat of the function: the squaring and summing. We will be abiding by the caller/callee convention, so in what register can we expect the parameter `n`? What registers should hold `square`'s parameter and return value? In what register should we place the return value of `sumSquare`?

```
        add  s0, a0, x0   # Set s0 equal to the parameter n
        add  s1, x0, x0   # Set s1 (accumulator) equal to 0
loop:   beq  s0, x0, end  # Branch if s0 reaches 0
        add  a0, s0, x0   # Set a0 to the value in s0, setting up
                          # args for call to function square
        jal  ra, square   # Call the function square
        add  s1, s1, a0   # Add the returned value into s1
        addi s0, s0, -1   # Decrement s0 by 1
        jal  x0, loop     # Jump back to the loop label
end:    add  a0, s1, x0   # Set a0 to s1 (desired return value)
```

6.2 Since `sumSquare` is the callee, we need to ensure that it is not overriding any registers that the caller may use. Given your implementation above, write a prologue and epilogue to account for the registers you used.

```
prologue: addi sp, sp -12   # Make space for 3 words on the stack
          sw   ra, 0(sp)    # Store the return address
          sw   s0, 4(sp)    # Store register s0
          sw   s1, 8(sp)    # Store register s1


epilogue: lw   ra, 0(sp)    # Restore ra
          lw   s0, 4(sp)    # Restore s0
          lw   s1, 8(sp)    # Restore s1
          addi sp, sp, 12   # Free space on the stack for the 3 words
          jr   ra           # Return to the caller
```

Note that `ra` is stored in the prologue and epilogue even though it is a call*er*-saved register. This is because if we call multiple functions within the body of `sumSquare`, we'd need to save `ra` to the stack on every call, which would be redundant — we might as well save it in the prologue and restore it in the epilogue along with the callee-saved registers. For this reason, in functions that don't call other functions, it is generally safe to refrain from saving/restoring `ra` in the prologue/epilogue as long as nothing else is overwriting it.