CS 61C Summer 2022 — RISC-V Discussion 4

1 Precheck

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 Translation

RISC-V is an assembly language, which is comprised of simple instructions that each do a single task such as addition or storing a chunk of data to memory.

For example, on the left is a line of C code and on the right is a chunk of RISC-V code that accomplishes the same thing.

```
int x = 5;
y[2];
y[0] = x;
y[1] = x * x;
```

```
// x -> s0, &y -> s1
addi s0, x0, 5
sw   s0, 0(s1)
mul  t0, s0, s0
sw   t0, 4(s1)
```

2.1 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:
```

3 Q3

In RISC-V, we have two methods of storing data: main memory and registers. Registers are much faster than using main memory, but are very limited in space (32 bits each). You should ALWAYS use the names of registers, e.g. `s0` rather than `x8`; the one exception to this rule is the zero register `x0`, as it is often shorter to write `x0` than its name `zero`, and the purpose of the register is still easy to tell with either identifier. The below table of register names is reproduced from the RISC-V green card.

Register(s)	Alt.	Description
x0	zero	The zero register, always zero
x1	ra	The return address register, stores where functions should return
x2	sp	The stack pointer, where the stack ends
x5-x7, x28-x31	t0-t6	The temporary registers
x8-x9, x18-x27	s0-s11	The saved registers
x10-x17	a0-a7	The argument registers, a0-a1 are also return value

3.1 Can you convert each instruction's registers to the other form?

```
add s0, zero, a1     -->          add x8, x0, x11
or  x18, x1, x30     -->          or  s2, ra, t5
```

As a reminder, you should ALWAYS use the named registers (e.g. `s0` rather than `x8`).

4 Q4

4.1 What is the range of 32-bit instructions that can be reached from the current PC using a branch instruction?

The immediate field of the branch instruction is 12 bits. This field only references addresses that are divisible by 2, so the immediate is multiplied by 2 before being added to the PC. Since it is signed, the branch immediate can therefore move the PC in the range of $[-2^{12}, 2^{12} - 2]$ bytes. If we're in a version of RISC-V that has 2-byte instructions, then this corresponds to a range of $[-2^{-11}, 2^{11} - 1]$ instructions. The instructions we use, however, are 4 bytes so they reside at addresses that are divisible by 4 not 2. Therefore, we can only reference half as many 4-byte instructions as 2-byte instructions, and the range of 4-byte instructions is $[-2^{10}, 2^{10} - 1]$

4.2 What is the maximum range of 32-bit instructions that can be reached from the current PC using a jump instruction?

The immediate field of the jal instruction is 20 bits, while that of the jalr instruction is only 12 bits, so jal can reach a wider range of instructions. Similar to above, this 20-bit immediate is multiplied by 2 and added to the PC to get the final address. Since the immediate is signed, we have a range of $[-2^{20}, 2^{20} - 2]$ bytes, or $[-2^{19}, 2^{19} - 1]$ 2-byte instructions. As we actually want the number of 4-byte instructions, we can reference those within $[-2^{18}, 2^{18} - 1]$ instructions of the current PC.

4.3 Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V green card!).

```
0x002cff00: loop: add t1, t2, t0          |_______|_______|_______|_______|_______|__0x33__|
0x002cff04:       jal ra, foo             |_____________________________|_____________|__0x6F__|
0x002cff08:       bne t1, zero, loop      |_______|_______|_______|_______|_______|__0x63__|
...
0x002cff2c: foo:  jr ra                  ra = ____________________________
```

```
0x002cff00: loop: add t1, t2, t0       |   0   |   5   |   7    |  0  |  6  |  0x33  | → 0x00538333
0x002cff04:       jal ra, foo          |  0  |  0x14  |  0  |   0   |   1   |  0x6F  | → 0x028000ef
0x002cff08:       bne t1, zero, loop   | 1 |  0x3F  | 0 | 6 | 1 |  0xC  | 1 |  0x63  | → 0xfe031ce3
...
0x002cff2c: foo:  jr ra                  ra = _________0x002cff08__________
```

4.1 What is the range of 32-bit instructions that can be reached from the current PC using a branch instruction?

The immediate field of the branch instruction is 12 bits. This field only references addresses that are divisible by 2, so the immediate is multiplied by 2 before being added to the PC. Since it is signed, the branch immediate can therefore move the PC in the range of $[-2^{12}, 2^{12} - 2]$ bytes. If we're in a version of RISC-V that has 2-byte instructions, then this corresponds to a range of $[-2^{-11}, 2^{11} - 1]$ instructions. The instructions we use, however, are 4 bytes so they reside at addresses that are divisible by 4 not 2. Therefore, we can only reference half as many 4-byte instructions as

2-byte instructions, and the range of 4-byte instructions is $[-2^{10}, 2^{10} - 1]$

4.2 What is the maximum range of 32-bit instructions that can be reached from the current PC using a jump instruction?

The immediate field of the jal instruction is 20 bits, while that of the jalr instruction is only 12 bits, so jal can reach a wider range of instructions. Similar to above, this 20-bit immediate is multiplied by 2 and added to the PC to get the final address. Since the immediate is signed, we have a range of $[-2^{20}, 2^{20} - 2]$ bytes, or $[-2^{19}, 2^{19} - 1]$ 2-byte instructions. As we actually want the number of 4-byte instructions, we can reference those within $[-2^{18}, 2^{18} - 1]$ instructions of the current PC.

4.3 Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V green card!).

```
0x002cff00: loop: add t1, t2, t0          |________|________|________|________|________|__0x33__|
0x002cff04:       jal ra, foo             |_________________________|_________________|__0x6F__|
0x002cff08:       bne t1, zero, loop      |________|________|________|________|________|__0x63__|
...
0x002cff2c: foo:  jr ra                 ra = _____________________________
```

```
0x002cff00: loop: add t1, t2, t0       |   0   |   5   |   7    |  0  |  6  |  0x33  | → 0x00538333
0x002cff04:       jal ra, foo          |  0  |  0x14  |  0  |   0   |   1   |  0x6F  | → 0x028000ef
0x002cff08:       bne t1, zero, loop   | 1 |  0x3F  | 0 | 6 | 1 |  0xC  | 1 |  0x63  | → 0xfe031ce3
...
0x002cff2c: foo:  jr ra                 ra = _________0x002cff08__________
```

5 Q5

5.1 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)
```

5.2 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.