UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC 170 Introduction to Computer Architecture Winter 2021

Quiz 3 Solutions

1. (12 points) Consider a new instruction for RISC-V (RV32I), `adds rd, rs1, rs2`. This instruction separately adds bits `<31:16>` of its two inputs and places the result in `rd<31:16>`, and also adds bits `<15:0>` of its two inputs and places the result in `rd<15:0>`. In English, this instruction separately adds the top 16 and bottom 16 bits of its two inputs.

   Write a RISC-V assembly program to implement `adds x2, x3, x4` using any RISC-V RV32I instructions of your choosing, and using `x5`–`x11` as temporaries if needed. Do not change `x3` and `x4`. You will help us grade and give partial credit if you comment your code enough for us to understand your intent. Your code does not have to be minimal but it does have to be near-minimal for full credit.

Solution:

Basic strategy is to separately compute the high and low parts of the answer, masking off the other bits, then `or` them together. Two solutions follow. Both are eight instructions.

One solution just uses `shift`s back and forth to zero out half the words, then `or`s the two half-results together. Note all shifts are logical so that we shift in only zeroes.

```
1 add x2, x3, x4   # full add, but we'll next throw away top 16 bits
2 sll x2, x2, 16   # shift left, bottom part of sum now in top part ...
3 srl x2, x2, 16   # ... and now it's back in the bottom part.
4 srl x5, x3, 16   # top half of x3 now in bottom half of x5
5 srl x6, x4, 16   # top half of x4 now in bottom half of x6
6 add x5, x5, x6   # add the top halves, result in bottom half
7 sll x5, x5, 16   # shift result back to the top ...
8 or  x2, x2, x5   # ... and or it into the final result
```

Same program showing bit values:

```
1                  # assume x3 = 0xabcdefgh, x4 = 0xijklmnop,
2                  #   and final solution = 0xqrstuvwx
```

```
add x2, x3, x4    #  x2: 0x????uvwx
sll x2, x2, 16    #  x2: 0xuvwx0000
srl x2, x2, 16    #  x2: 0x0000uvwx
srl x5, x3, 16    #  x5: 0x0000abcd
srl x6, x4, 16    #  x6: 0x0000ijkl
add x5, x5, x6    #  x5: 0x????qrst
sll x5, x5, 16    #  x5: 0xqrst0000
or  x2, x2, x5    #  x2: 0xqrstuvwx
```

Alternately, we can use bitmasks. There's probably several ways to generate a bitmask that covers either the top half or bottom half of the word, but by far the easiest is `lui`. We then do a full `add` and throw away the top 16 bits of the result (to calculate the bottom half of the sum) and then mask off the bottom 16 bits of each input and `add` (to calculate the top half of the sum), then `or` them together.

```
lui x6, 0xffff0   # x5 now has 0xffff0000
xori x6, x5, -1   # x6 now has 0x0000ffff
add x7, x3, x4    # full add, but we'll next throw away top 16 bits ...
and x2, x6, x7    # ... and put it in x2
and x8, x5, x3    # set bottom 16 bits of x3 to 0
and x9, x5, x4    # set bottom 16 bits of x4 to 0
add x7, x8, x9    # full add of just top 16 bits ...
or x2, x2, x7     # ... and or it into the final result
```


Same program showing bit values:

```
                  # assume x3 = 0xabcdefgh, x4 = 0xijklmnop,
                  #   and final solution = 0xqrstuvwx
lui x5, 0xffff0   # x5: 0xffff0000
xori x6, x5, -1   # x6: 0x0000ffff
add x7, x3, x4    # x7: 0x????uvwx
and x2, x6, x7    # x2: 0x0000uvwx
and x8, x5, x3    # x8: 0xabcd0000
and x9, x5, x4    # x9: 0xijkl0000
add x7, x8, x9    # x7: 0xqrst0000
or x2, x2, x7     # x2: 0xqrstuvwx
```

2. *To make this problem simpler, all registers in this problem are 32 bits.*

Recall that a 32-bit IEEE 754 single-precision floating point value (referred to in the rest of this problem as a "float") has a sign field (bit 31), an exponent field (bits 30–23), and a fraction field (bits 22–0).

Eddie is thinking of two 32-bit values that he can place in integer register `x2`. Each of these values, when interpreted as a float, has the following property: Its value can also be exactly represented as a *positive (non-zero) 32-bit unsigned integer*. (As an example, the float 42.0 can be represented exactly as a 32-bit unsigned integer—`0x2a`—but the float 6.02×10^{23} cannot.)

(a) (4 points) Eddie's first value is the smallest number that satisfies this property. **Write the hexadecimal value of the floating-point representation of this number.** (In other words, what are the 32 bits that would be stored in `x2` that represent this number?)

> **Solution:** This number is 1.0, which has a 0 sign bit, 127 in the exponent field, and 0 in the fraction field. Thus **0x3f800000**.

(b) (8 points) Eddie's second number is the largest number that satisfies this property. **What is the exponent field of this number?** (Your answer should be an unsigned integer between 0–255.) You do not need to find the exact number to determine the exponent. Note: the largest expressible float is much larger than the largest expressible unsigned integer.

> **Solution:** In this solution, the "exponent" is the actual power of 2 for the float and the "exponent field" is the actual bits in the float representation (your answer). Remember, for single-precision floating point, exponent + 127 = exponent field.
>
> The largest possible unsigned integer will have a 1 in bit 31. Let's first look at a number with exponent=0 (so the exponent field has 127). This number is between 1 and 2, so its highest order bit is in bit position 0. So we need to move this bit 31 places to the left (multiply it by 2^{31}), so exponent = 31 and **exponent field** = 127 + 31 = 158.

3. (4 points) In class we discussed that the RISC-V RV64I instruction `beq rs1, rs2, target` ("branch if equal") is (likely) implemented in the ALU by subtracting its two arguments (`rs1 - rs2`) and branching if the result is zero. Calculating the branch condition, then, requires a large `nor` gate on the ALU output `out`:

   `branch_condition = nor(out<63>, out<62>, ..., out<0>).`

   Assume that the RISC-V RV64I instruction `bge rs1, rs2, target` ("branch if greater than or equal") is also implemented by subtracting its two operands (`rs1 - rs2`). **Write the minimal logic equation for the `bge` branch condition as a function of the 64-bit `out` output of the ALU.** Express your answer as

   `branch_condition = ...`

Solution: If `rs1 - rs2` is positive or zero, the branch condition is also positive. We can determine this by looking at the sign bit, which is zero, so we have to invert it to indicate the branch condition. So:

`branch_condition = not(out<63>)`.