

## ECE 6913, Computing Systems Architecture, Fall 2019

### *Quiz 1, October 9<sup>th</sup> 2019*

Maximum time: 2.5 hours : **9:50 AM – 12:20 PM**

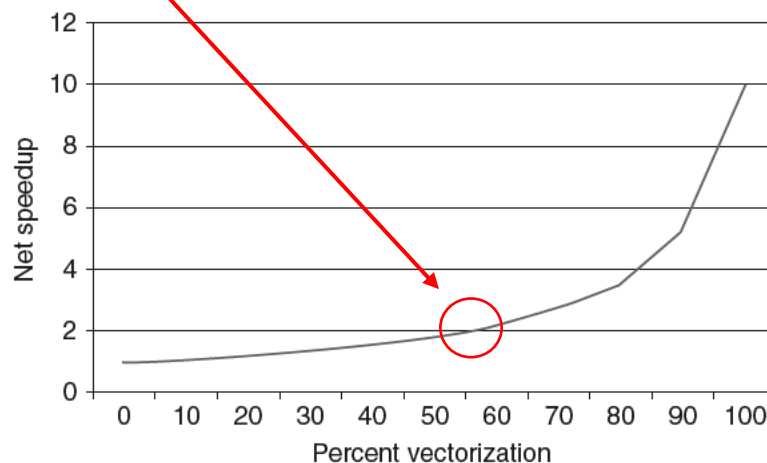
*Closed Book, Closed Notes, No 'cheat sheet' allowed*

*Calculators allowed. RISC V Card provided*

This Test has 5 problems each with several parts. Please attempt all of them. Please show all work. Please write legibly

1. In this exercise, assume that we are considering enhancing a machine by adding vector hardware to it. When a computation is run in vector mode on the vector hardware, it is **10 times faster than the normal mode of execution**. We call the percentage of time that could be spent using vector mode the *percentage of vectorization*. Vectors are discussed in Chapter 4, but you don't need to know anything about how they work to answer this question!

**x = 55.55%**



**Plot of the equation:  $y = 100 / [(100-x) + x/10]$**

- a. Approximately plot the speedup as a percentage of the computation performed in vector mode by identifying a few data points on the plot. Label the y-axis "Net speedup" and label the x-axis "Percent vectorization."

Assume  $T_0$  is the time taken for the non vectorized code to run.

assume 'x' equals the percentage of the code vectorized. This piece takes  $1/10$  the time to run as the non-vectorized code. So,  $[100-x]\%$  of the code is not vectorized.

total time to run code with x % vectorized =  $[100 - x]\% T_0 + [x/10]\%T_0$

if  $x=0\%$ , then the code takes 100% of  $T_0$  time to execute

if  $x=100\%$ , then code takes 10% of  $T_0$  time to execute

$$\text{Speedup} = 100\%T_0 / \{ [100 - x]\% T_0 + [x/10]\%T_0 \}$$

- b. *What percentage of vectorization is needed to achieve a speedup of 2?*

or speedup of 2 (code runs twice as fast, that is, takes half as long

$$2 = 100\%T_0 / \{ [100 - x]\% T_0 + [x/10]\%T_0 \}$$

$$2 = 100 / \{ [100 - x] + [x/10] \}$$

- c. *What percentage of the computation run time is spent in vector mode if a speedup of 2 is achieved?*

$$[x/10] / [ (x/10) + (100-x) ] = 5.55 / [ 5.55 + 44.45 ]$$

$$= 5.55/50$$

$$= 0.111 \times 100 = 11.11\%$$

- d. *What percentage of vectorization is needed to achieve one-half the maximum speedup attainable from using vector mode?*

maximum speedup from using vector mode = 10x

half of maximum speedup = 5x

$$5 = 100\%T_0 / \{ [100 - x]\% T_0 + [x/10]\%T_0 \}$$

$$\text{So, } x = 88.88\%$$

i.e., 88.88% of code must be vectorized for speedup to be half of maximum:

that is  $= 1/2 \times 10 = 5$

2. Find a short sequence of RISC-V instructions that extracts bits 23 down to 8 from register x7 and uses the bitwise inverted value of this field to replace bits 31 down to 16 in register x8 without changing any of the bits of registers x7 or any of the bits in the less significant half word of x8 (bits 15:0).

```
lui x5 0x0ffff          // fill most significant 20 bits (5 hex) with
                        // 0ffff char: x5 now has these 8 hex:
                        // 0ffff000
slri x5 8                // x5 now has 000ffff0 with bits 23 to 8
                        // now a '1'

and x28 x5 x7            // apply mask to x7 and place 16 desired
                        // bits in positions 23 → 8 from x7 into x28
                        // all bits in positions other than 23→8
                        // are '0'

xori x29 x28 -1          // invert these bits and update them in x29
and x28 x5 x29           // copy only these inverted bits back to x28
                        // using the original mask identifying
                        // positions 23 → 8. all other bits are '0'

slli x5 x5 8             // create mask for x8 to identify 16 bits in
                        // most significant 16 bit positions

xori x5 x5 -1            // invert this mask placing '0' 16 most
                        // significant positions and '1'
                        // in least significant positions

and x8 x8 x5            // zero out the most significant 16
                        // positions in x8

slli x28 x28 8           // 16 desired bits from positions 23→8 in x7
                        // are now moved into most
                        // significant positions 31 - 16 in x28

or x8 x8 x28            // simply copy 16 desired bits from x7 into
                        // most significant 16 positions in x8
```

3. One possible performance enhancement is to do a shift and add instead of an actual multiplication. Since  $9 \times 6$ , for example, can be written  $(2 \times 2 \times 2 + 1 \times 6)$ , we can calculate  $9 \times 6$  by shifting 6 to the left three times and then adding 6 to that result. Show the best way to calculate  $0 \times 3A_{\text{hex}} \times 0 \times 5F_{\text{hex}}$  using shifts and adds/subtracts. Assume both inputs are 8-bit unsigned integers.

$$0 \times 3A_{16} \times 0 \times 5F_{16} = 0 \times 1586_{16}.$$

$$0 \times 3A_{16} = 58, \text{ and } 58 = 32 + 16 + 8 + 2$$

We can shift  $0 \times 5F$  left 5 places ( $BD0_{16}$ ), then add  $0 \times 5F$  shifted left 4 places ( $5F0_{16}$ ), then add  $0 \times 5F$  shifted left 3 places ( $2F8_{16}$ ) and then add  $0 \times 5F$  shifted once ( $0BD_{16}$ )

$$5F \times 2^5$$

$$= 5F \text{ shifted left 5 places: } 0101 \ 1111 \text{ shifted left 5 places} = 1011 \ 1110 \ 0000 = BD0_{16}$$

$$5F \times 2^4$$

$$= 5F \text{ shifted left 4 places: } 0101 \ 1111 \text{ shifted left 4 places} = 0101 \ 1111 \ 0000 = 5F0_{16}$$

$$5F \times 2^3$$

$$= 5F \text{ shifted left 3 places: } 0101 \ 1111 \text{ shifted left 3 places} = 0010 \ 1111 \ 1000 = 2F8_{16}$$

$$5F \times 2^1$$

$$= 5F \text{ shifted left 1 places: } 0101 \ 1111 \text{ shifted left 1 places} = 0000 \ 1011 \ 1110 = 0BD_{16}$$

$$BD0 + 5F0 + 2F8 + 0BD = 0 \times 1586_{16}.$$

4 shifts, 3 adds.

3. Write the RISC-V assembly code that creates the 64-bit constant  $0x1122334455667788_{16}$  and stores that value to register x10.

```
lui x10, 0x11223          // loads the 5 hex numbers 11223 into the upper 20 bits of x10
                           // note that each hex number corresponds to 4 bits, so loading the
                           // upper 20 bits with lui can only permit exactly 5 hex numbers.
                           // Hence the choice of '11223' into the upper 20 bits
addi x10, x10, 0x344      // we can extend the above string of 11223 by adding the immediate
                           // value of 3 hex numbers of '344' in the lower 12 bits to the
                           // above string of '11223' to get '11223344' into x10
slli x10, x10, 32         // moves the 8 hex numbers '11223344' to the upper half of the 64
                           // bit double word
lui x5, 0x55667          // loads the MSBs of the lower half of the desired double word with
                           // the 5 hex numbers '55667'
addi x5, x5, 0x788        // just as we did previously, these 5 hex numbers are added to 3
                           // hex numbers '788' to produce the lower half string: '55667788'
add x10, x10, x5          // this added to the upper half string yields the full desired
                           // string of '1122334455667788' loaded into the double word
```

4. IEEE 754-2008 contains a half precision that is only 16 bits wide. The leftmost bit is still the sign bit, the exponent is 5 bits wide and has a bias of 15, and the mantissa is 10 bits long. A hidden 1 is assumed.

(a) Write down the bit pattern to represent  $-1.5625 \times 10^{-1}$  assuming a version of this format, which uses an excess-16 format to store the exponent. Comment on how the range and accuracy of this 16-bit floating point format compares to the single precision IEEE 754 standard.

$$-0.15625_{10} = -0.00101_2 = -1.01 \times 2^{-3}$$

Sign bit = 1

10 bit Fractional field = 0100000000

5 bit Exponential Field = val(exponent) + Bias =  $-3 + 15 = 12 = 01100$

So, 16 bit IEEE 754-2008 representation of  $-0.15625$ :

1 01100 0100000000

- (b) The associative law for addition says that  $a + (b + c) = (a + b) + c$ . This holds for regular arithmetic, but does not always hold for floating-point numbers. Using the 16-bit floating-point representation, give an example of three floating-point numbers  $a$ ,  $b$ , and  $c$  for which the associative law does not hold, and show why the law does not hold for those three numbers.

Let

$a = 1 \ 11110 \ 1111111111$

$b = 0 \ 11110 \ 1111111111$

Let  $c = 1.0_2 \times 2^{-15}$

$$-15 = \text{EXP} - 15 \Rightarrow \text{EXP} = 0$$

$c = 0 \ 00000 \ 0000000000$

$c$  is outside of the range that can be represented by a 16 bit FP standard

Now,  $a$  and  $b$  cancel each other (one is negative of the other)

Then  $(a + b) + c = c$ , because  $a$  and  $b$  cancel each other, however,  
 $a + (b + c) = 0$ , because  $b + c = b$  ( $c$  is very small relative to  $b$  and is lost in underflow).