1.

What are the differences between RISC and CISC? What are some of the advantages and disadvantages?

RISC: Reduced instruction set

* Individual instructions are simpler, meaning more instructions are required to do a certain task (on average)
* More parallelism and pipelining allowed because a more complex instruction would be broken up into smaller instructions and the dependencies are more relieved
* Takes up more memory because more instructions are required
* Faster decoding due to constant instruction lengths (32 bit)
* More registers to store more values at a certain time – less chance of register spilling
* 1 clock cycle per instruction
* Fewer transistors used for instructions

CISC: Complex instruction set

* Less memory required because a single instruction can do more things and can replace multiple instructions in a RISC architecture
* Less opportunity for pipelining
* Multiple clock cycles per instruction
* More chip space required for instructions

2.

Translate the following x86 instructions into MIPS:

a.

add 0x200(,%rdx,4),%rcx

Assume $t0 corresponds to %rdx and $t2 corresponds to %rcx

sll $t0, $t0, 2

addi $t0, $t0, 0x200

lw $t1, 0($t0)

add $t1, $t1, $t2

b.

lea 0xc(%rdi),%rax

Assume $t0 corresponds to %rdi and $t1 corresponds to %rax

addi $t1, $t0, 0xc

c.

mov 0x30(%rsp,%rbx,4),%rax

Assume $t0 corresponds to %rbx, $t1 corresponds to %rsp, and $t2 corresponds to %rax

sll $t0, $t0, 2

add $t1, $t1, $t0

lw $t2, 0x30($t1)

d.

mov %rcx,-0x30(%rsp,%rdx,4)

sll $t0, $t0, 2

add $t0, $sp, $t0

sw $t2, -0x30($t0)

3.

Translate the x86 code into MIPS. Assume variables a,b, and i are in register $s0, $s1, and $t0. Assume a, b, and i are in rdi, rsi, and rdx.

for(i = 0; i < 5; i++) {

a+=b;

}

mov $0, rdx

.loop: cmp $4, rdx

jg leaveloop

add rsi, rdi

add $1, rdx

jmp .loop

li $t0, 0

loop:

slti $t2, $t0, 0x5

beq $t2, $zero, done

add $s0, $s0, $s1

addi $t0, $t0, 0x1

j loop

done:

4.

What does the following MIPS code snippet do?

Loop: lw $t0, 0($s0)

lw $t1, 0($t0)

add $t1, $s1, $t1

sw $t1, 0($t0)

addi $s0, $s0, 4

bne $s0, $s2, Loop

Thought process: The first two instructions will load an integer from memory into $t1. $s0 is a pointer to a pointer to the integer, and $t1 will then have the dereferenced pointer, which is a pointer still. Then, that pointer is dereferenced once more to be stored in $t1. That integer is then added to a different variable, then stored back in the original pointer. The original pointer is now pointing at the element in memory 4 bytes down, meaning the loop is probably traversing through an array of integers, until $s0 == $s2, where $s2 is likely a pointer to the end of the array.

Overall: $s0 is a pointer to an array of pointers to some data type of 4 bytes. Each element pointed to by the array, starting at wherever $s0 is pointing is incremented by $s1, until reaching the part of the array pointed to by $s2.

5.

When does False Sharing occur, and how does it affect performance when parallelizing?

False sharing occurs when two threads share the same cache line/block, and when one thread modifies a value in that cache line. Even if the other thread doesn’t modify the value, both threads must refresh that value by pulling it from memory again. In doing this, the caches maintain cache coherency, as the changes in one cache are “coherent” in the other. It slows down performance when parallelizing because these unnecessary access to memory must be made. This is called false sharing because each thread is not actually sharing access to the same variable.