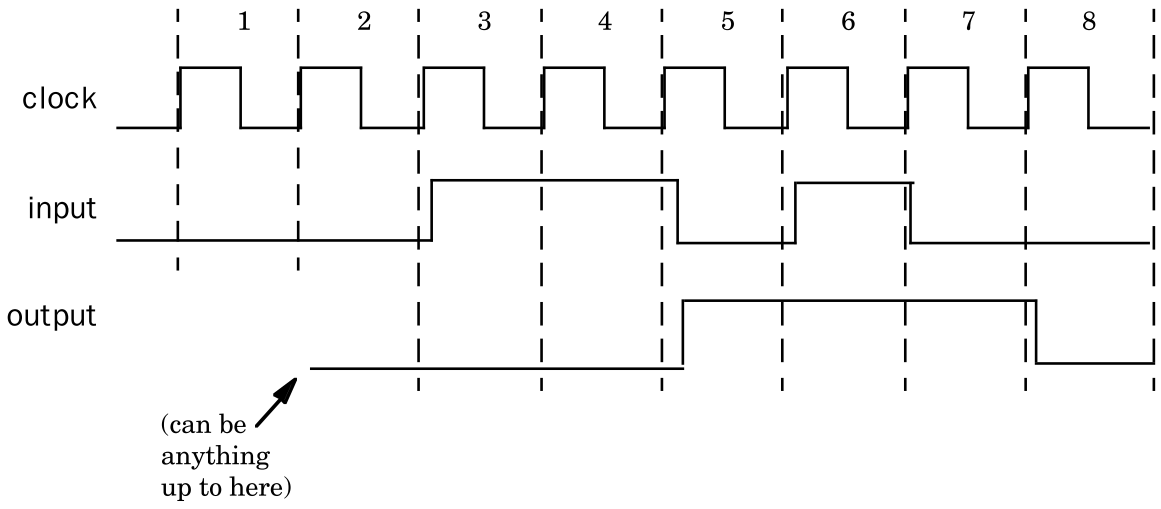
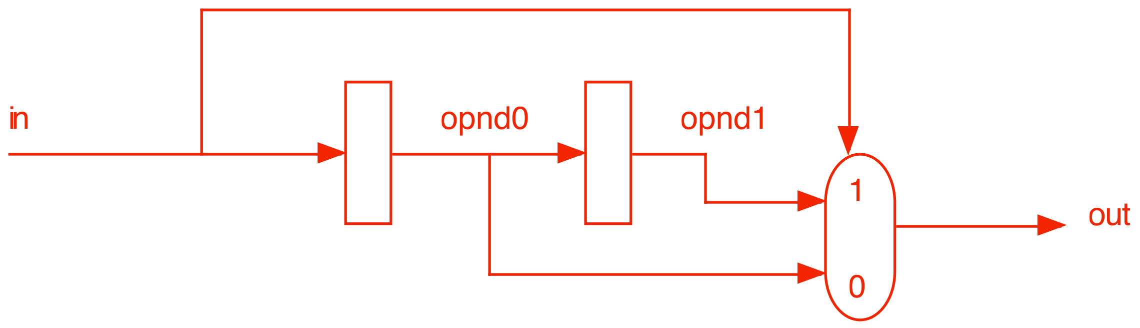
Problem 1 (logic design)

For this problem, you are to design and implement a sequential multiplexor that works as follows. On each clock cycle, interpret the current input as a selector from the most recent input (operand 0) and the input before that (operand 1), and output the result. All signals are one bit each. Here is a timing diagram.



Draw a schematic (a logic diagram) for your implementation.

You may wish to add timing waveforms for signals inside your circuit to the above diagram. You may also use a state transition diagram. Neither of these will be scored, but they may prove useful in designing and visualizing a solution. [Note that timing diagrams were not covered in CS 61CL this semester.]



Problem 2 (CPU design)

The RISC approach is carried to an extreme in the Single Instruction Computer (SIC). This computer has no registers and only one instruction:

```
sbn addr1, addr2, jumpAddr (Subtract and Branch if Negative)
```

Given three memory addresses addr1, addr2, and jumpAddr, this instruction subtracts the contents of addr2 from the contents of addr1, stores it into the location addressed by addr1, and if the result of the subtraction is negative, jumps to the instruction addressed by jumpAddr. In more concise notation,

```
Mem[addr1] = Mem[addr1] - Mem[addr2];
if (Mem[addr1] < 0) go to jumpAddr;
```

The SIC program segment shown below copies a number from location a to location b. It assumes that temp labels a spare memory word that can be used for temporary results.

```
      .text
copy:
      sbn temp,temp,label1 # sets temp to 0; does not branch
label1:
      sbn temp,a,label2    # sets temp to -a;
      # continues with next instruction regardless of a's sign
label2:
      sbn b,b,label3       # sets b to 0; does not branch
label3:
      sbn b,temp,label4    # sets b to -temp, which is a;
      # continues with next instruction regardless of b's sign
label4:
      ...
      .data
temp:
      .word 0
```

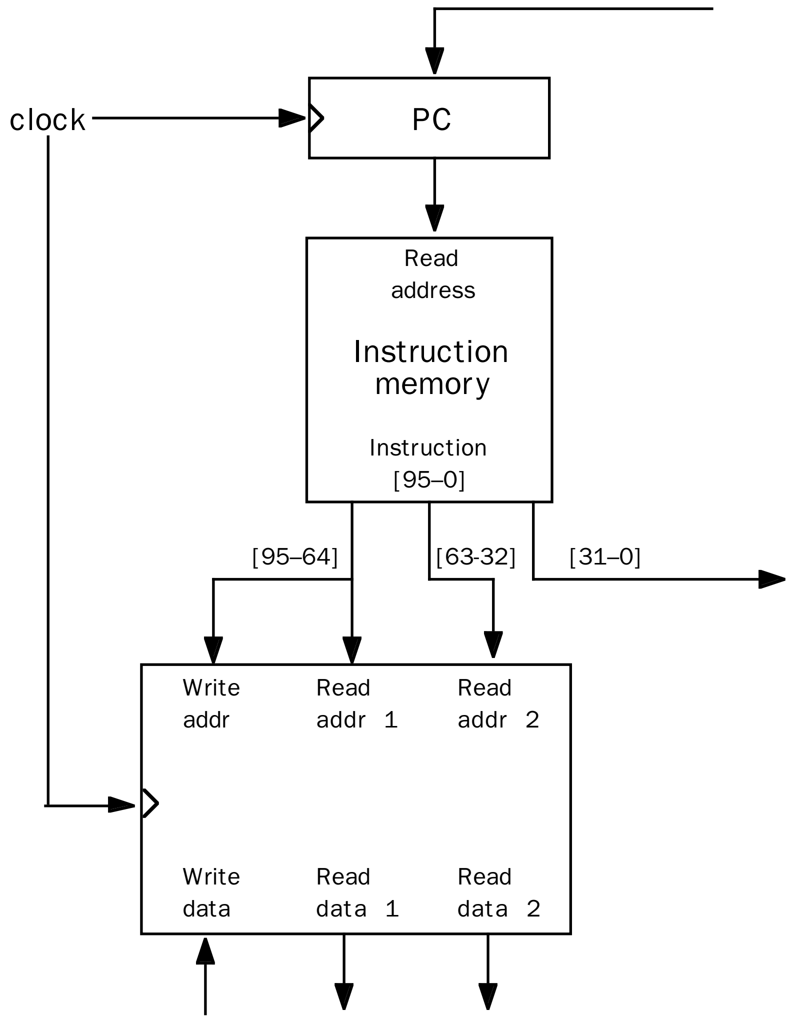
Now consider the instruction encoding and implementation of this computer. Since there's only one instruction, there's no need for an op code. The format of an SIC instruction is merely three 32-bit addresses as shown below: bits 95-64 represent addr1, bits 63-32 represent addr2, and bits 31-0 represent jumpAddr.

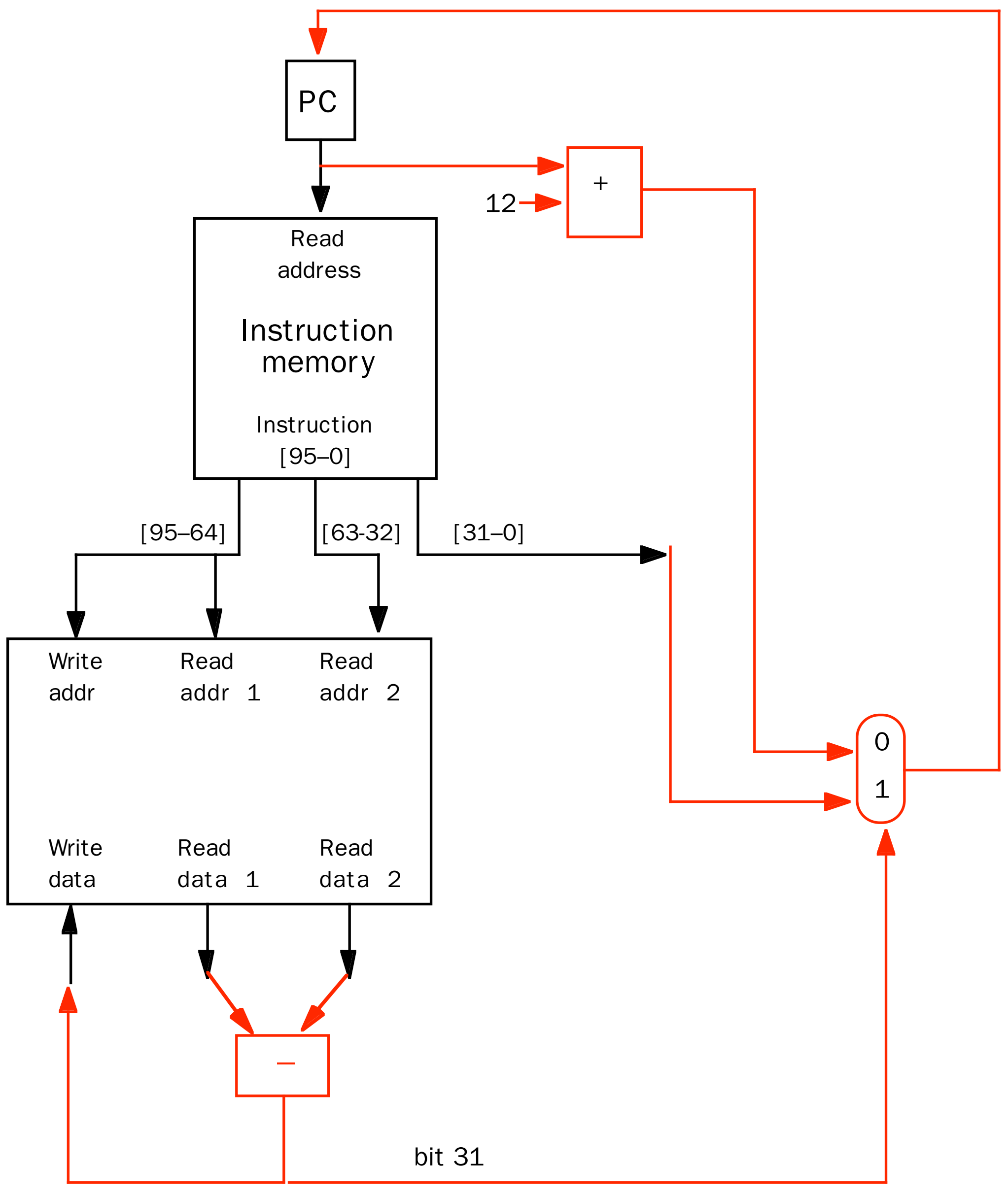
95 64	63 32	31 0
addr1	addr2	jumpAddr

One instruction is executed per clock cycle. Instruction memory is separate from data memory. Assume also that a special data memory is used that can read and write multiple values in the same clock cycle.

Part a

Complete the diagram of the circuitry of the SIC CPU below.





Part b
Explain at least one aspect of this architecture that would make it hard to pipeline.

There are several operations involved in executing an instruction:
1. instruction fetch;
2. data fetch (two arguments simultaneously);
3. subtraction;
4. determining whether to branch.

Each step depends on the previous one; moreover, the PC update is the last thing to happen, which would cause one branch delay for each stage of the pipeline after the first. This is complicated by the fact that *every* instruction might involve a branch.

Another complication involves potential contention for the memory bus.

Problem 3 (CPU design)

Consider the addition of the max instruction to the MIPS instruction set:

```
        max   Rdest, Rsource1, Rsource2
```

It stores the larger of the values in registers Rsource1 and Rsource2 into register Rdest.

Part a

Design a machine representation for the max instruction that's consistent with the existing MIPS instructions. Clearly indicate the purpose of each bit field in the instruction, using the format of Patterson and Hennessy appendix A.

This part was worth 3 points, distributed as follows:

a. This is an R-format instruction, so the opcode is 0.

b. It uses the ALU, so function code should be something unused and grouped with the other ALU-using instructions (which start with the add instruction at 0x20). We gave credit for 0x28 (decimal 40), 0x29 (decimal 41), or anything greater than or equal to 0x2c (decimal 44)and at most 0x3F.

c. Operand format should be consistent with the other instructions and with the format of P&H appendix A. In particular, the shamt field is 0. Here's a full-credit description.

    000000 sssss ttttt ddddd 00000 101000   max Rd,Rs,Rt

1 point was deducted per error. Common errors were the choice of a function code that was too small or equal to an existing function code, misordering the Rd, Rs, and Rt fields, and neglecting to specify a shamt field of 0. Some students also concluded incorrectly from part b that they were not allowed to use a 0 opcode. (A Max signal provided by the "instruction decoder" can come from ALU control, which does instruction decoding.)

To be strictly consistent with the other instructions, the function code should match the circuitry described on page 356 (figure 5.15). This would be 0x3A (decimal 58) for a max implementation based on SLT, or 0x32 (decimal 50) for an implementation based on SUB.

Part b

Indicate by descriptions below and by additions to Figure 5.19 what changes to the datapath are necessary and what values existing signals must take on to implement the max instruction. Briefly explain your answers. You may assume that a new Max signal is provided by the instruction decoder. Your changes should not involve changing the ALU or adding a new ALU.

Values of existing signals:

Max = 1	Branch =	
MemRead =	MemtoReg =	MemWrite =
RegDst =	RegWrite =	

ALUSrc =	ALUOp =	
ALU control =		

Brief explanation of control signal values:

RegDst = 1 since R-format
Branch = 0, MemRead = MemtoReg = MemWrite = 0
ALUSrc = 0 since R-format
RegWrite = 1
ALUOp = 10 since R-format
ALU control = 111 (slt) or 110 (sub)

Other changes:

Read data 1 (0) and Read data 2 (1) should be routed to a Mux whose control signal is ALU_result. That result should be sent to the 0 input of the rightmost Mux (you need another Mux to do this, with Max as the control signal).

Another approach is to use the top bit of the result of subtracting the second register from the first as the control signal for the first multiplexor.

Problem 4 (caches)

Part a

Consider a 4-kilobyte direct-mapped cache with a block size of 2 words. Indicate below which bits of a 32-bit address form the tag, which form the cache index, and which form the byte offset (the position in the block).

byte offset = bits 2-0 (3 bits), cache index = bits 11-3 (9 bits), tag = bits 31-12 (20 bits)

Part b

Consider now a 8-word direct-mapped cache with 2-word blocks, and suppose that the following sequence of memory accesses is made (e.g. with a sequence of loads) with an initially empty cache. Identify which accesses are hits, which are misses that fill in a block, and which are misses that cause a block to be replaced, by marking each with H, M, or MR.

hex byte address	hit (H), miss (M), or miss with replacement (MR)?
4512	
4514	
4504	
4501	
4508	
4584	
4518	
4501	

Byte offset is 3 bits, index is 2.

4512 ends in 10010 (first word in block); miss; cache is now

0	-- --- ---
1	-- --- ---
2	45+000 10000-10111
3	-- --- ---

4514 ends in 10100 (second word in block); hit

4504 ends in 00100 (second word in block); miss; cache is now

0	45+000 00000-00111
1	-- --- ---
2	45+000 10000-10111
3	-- --- ---

4501 ends in 00001 (first word in block); hit

4508 ends in 01000 (first word in block); miss; cache is now

0	45+000 00000-00111
1	45+000 01000-01111
2	45+000 10000-10111
3	-- --- ---

4584 ends in 00100 (second word in block); miss with replacement; cache is now

0	45+100 00000-00111
1	45+000 01000-01111

2	45+000 10000-10111
3	-- --- ---

4518 ends in 11000 (first word in block); miss; cache is now

0	45+100 00000-00111
1	45+000 01000-01111
2	45+000 10000-10111
3	45+000 11000-11111

4501 ends in 00001 (first word in block); miss with replacement; cache is now

0	45+000 00000-00111
1	45+000 01000-01111
2	45+000 10000-10111
3	45+000 11000-11111

Problem 5 (caches)

Consider a 16-word (not counting tags) 2-way associative cache with a block size of 4 words using LRU replacement.

Part a
Indicate which bits of a 32-bit address form the tag, the cache index, and the byte offset.

byte offset = bits 3-0, cache index = bit 4, tag = bits 31-5

Part b
Suppose that the contents of memory between byte addresses 52 and 83 are as shown below.

byte address	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67
contents	3				1				4				5			

byte address	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83
contents	9				2				8				6			

In the diagram below, fill in the result of loading the word at address 68, using a cache that's initially empty.

set	data (contents)			
0				
1				

Words at addresses 64, 68, 72, and 76 are loaded, so contents are 5, 9, 2, 8 in set 0.

Part c
The cache.c program from homework 9, run on a computer with the cache just described (and no secondary cache), produces a "read+write" time of 100ns in the situation where the number of cache hits is maximized and a time of 900ns when the number of cache hits is minimized.

By filling in the bottom row of the table below, indicate what times this run of cache.c might produce for a 32-word array with strides ranging from 1 to 16 words. Each value will be one of the following: 100ns, 300ns, 500ns, 900ns.

		stride in words	1	2	4	8	16
		stride in bytes	4	8	16	32	64
size in words	bytes						
8	32		100	100	100		
16	64		100	100	100	100	

32 128					

In a 16-word array with a stride of 1 word, there are misses only when array elements 0, 4, 8, and 12 are accessed for the first time. If each word in the array is accessed 10 times, this is only 4 misses out of 160 accesses.

In a 32-word array with a stride of 1 word, there are misses each time elements 0, 4, 8, 12, 16, 20, 24, 28 are accessed. This is 80 misses out of 320 accesses.

A stride of 2 words produces 80 misses out of 160 accesses.

A stride of 4 words produces all misses.

A stride of 8 words accesses elements 0, 8, 16, 24. These all map to row 0 of the cache, and thus also produces all misses.

A stride of 16 words accesses elements 0 and 16, which can both be stored in row 0 of the cache. This produces all hits.

Thus the entries are 300, 500, 900, 900, and 100.

Problem 6 (K&R storage management code)

Suppose that a programmer using the K&R storage allocator accidentally overwrites the size of an allocated block—we'll call it B—with a 0.
What will be the effect of this accident?

- The accident won't cause any problem at all.
- The accident will cause a crash when the overwrite occurs.
- The accident will cause a crash when block B is freed.
- The accident may cause a crash somewhere after the overwrite, but not necessarily when block B is freed.
- The accident won't cause a crash, but it will produce some storage that can no longer be used (a memory leak).

Briefly explain your answer.

Choice e, memory leak.
Overwriting the size with zero will just make a block that looks like a block of size 0. The block will never be reallocated after it's freed; there won't be a problem combining it with a neighboring block; and the remainder of the block will never be used for anything.

Problem 7 (virtual memory)

For this problem, make the following assumptions.

- The TLB is fully associative, and holds four entries.
- The program counter contains 0x00408940.
- The instruction at virtual address 0x00408940 is lw $t1, 0($t0).
- Register $t0 contains 0x1002A128.
- The page size is 8K.
- The address of the page table is kept in a special hardware register (so it doesn't need to be kept in the TLB).
- The TLB is currently empty.
- The page table contains the following valid entries:

virtual page number	physical page number
0x000102	0x023
0x000204	0xFF4
0x000408	0x038
0x00400A	0xFEA
0x004089	0xABC
0x008015	0x5BC
0x01002A	0x891
0x1002A1	0x007

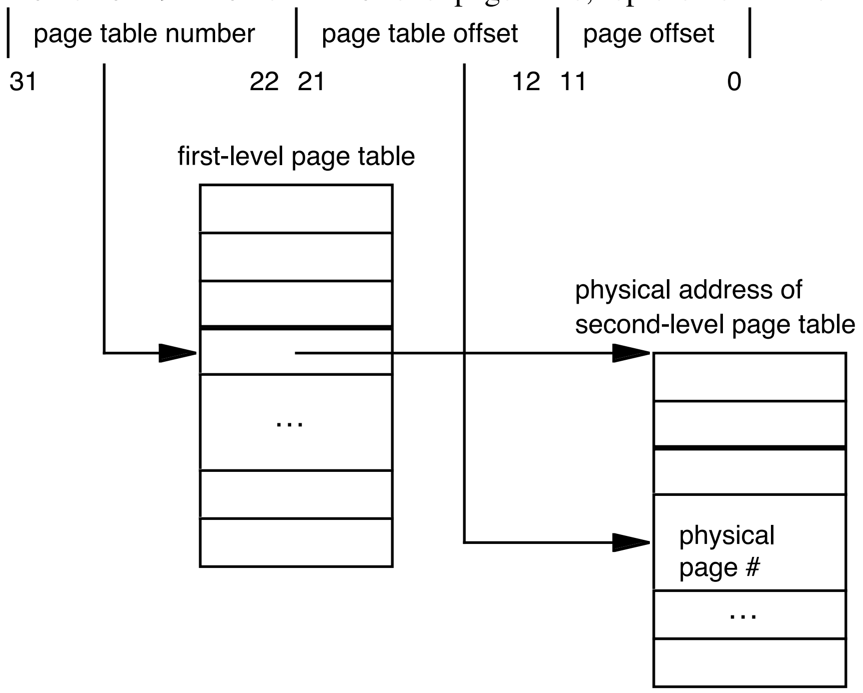
What does the TLB contain after execution of the lw instruction at address 0x00408940?

We assume that virtual addresses are 32 bits. Since the page size is 8K, the page offset is 13 bits and the virtual page number is 19 bits.

The instruction fetch is done first. The VPN is the first 19 bits of the address 0x00408940, namely 0x00204 (listed with an irrelevant most significant 0 in the table), so the pair (0x00204, 0xFF4) goes into the TLB first. Then the data reference is made to the address 0x1002A128. Its VPN is 0x08015 (again listed with an irrelevant 0), and the pair (0x008015, 0x5BC) goes into the TLB.

Problem 8 (virtual memory)

Homework 9 involved a two-level page table, represented in the diagram below.



For the homework, we assumed that the length of a virtual address was 32 bits. List three different ways of accommodating a 33-bit virtual address in this address translation system while retaining the two-level table structure and 32-bit physical address length. For each method you list, describe its specific effect on page sizes and the page tables.

1. Double the size of the primary page table; leave everything else the same.
2. Double the size of the secondary page tables; leave everything else the same.
3. Double the size of pages, but leave the page tables unchanged.

Problem 9 (virtual memory)

Suppose that a MIPS computer has a fully associative TLB, a 32-bit virtual address space, and a page size of 4KB. Suppose also that a suspended process gets swapped in, the TLB gets emptied, and the process resumes execution at virtual address 404FA000. Finally, suppose that after a small number of instructions, the program has made no references to data memory, and the TLB contains two valid entries:

tag	physical page address
404FA	00272
08A1E	00138

What MIPS instructions and pseudoinstructions were executed to produce the above TLB contents? List each one below (in assembly language format) along with its virtual address.

Since no references to data memory are made, the instructions must be either jumps, branches, R-format instructions, or loads from or stores into the text segment.

Starting at virtual address 404FA000:

 la $t0,label (this is instructions 404FA000 and 404FA004)
 jr $t0 (at 404FA008)

Starting at virtual address 08A1E000:

 (any instruction)

Note that the top two bits of the source and the destination of the jump differ, so we need to use la and jr. An assembler, however, would convert a long jump to that sequence of instructions.

Problem 10 (i/o)

The MIPS assembly program on the next page reads exactly 1024 keystrokes from the keyboard and sends them over a modem in any order. It also performs “other work” while waiting for data.

The keyboard

The code at intHandler is the interrupt handler for the keyboard. When a key is pressed on the keyboard, it triggers the interrupt. The interrupt handler is then invoked and should then read the character corresponding to the pressed key from address 0x00FF0000.

The modem

The modem’s status can be read from address 0x00FA0000; if this value is zero, the modem is ready to accept data. Data is sent to the modem by writing to address 0x00FA0004.

The buffer

Since the order of the keystrokes does not need to be preserved, the programmer has chosen to store them in a stack at address stack. The word at address stackTop contains the address of the top of the stack.

```

stack:      .space      1024
stackTop:   .word stack

intHandler:
la    $k0,0x00FF0000    # $k0 = address of keyboard data
lb    $k0,0($k0)  # $k0 = the keystroke
la    $k1,stackTop      # $k1 = address of stackTop variable
lw    $k1,0($k1)  # $k1 = address of top of stack
sb    $k0,0($k1)  # mem[stackTop] = newly-read keystroke
addi  $k1,$k1, 1  # $k1 = new top of stack
la    $k0,stackTop      # $k0 = address of stackTop variable
sw    $k1,0($k0)  # stackTop = $k1
eret

main:
la    $t0,0x00FA0000    # $t0 = address of modem status
lw    $t1,0($t0)  # $t1 = modem status
bne   $t1,$0,otherWork  # if modem not ready, do other stuff

la    $t0,stackTop      # $t0 = address of stackTop
lw    $t1,0($t0)  # $t1 = address of top of stack
la    $t2,stack   # $t2 = address of bottom of stack
beq   $t2,$t1,otherWork # if stack top and bottom same, jump

addi  $t1,$t1,-1  # $t1 = new top of stack
lb    $t3,0($t1)  # $t3 = key at top of stack
sw    $t1,0($t0)  # mem[stackTop] = new top of stack
la    $t0,0x00FA0004    # $t0 = address of modem output
sb    $t3,0($t0)  # modem output = $t3

otherWork:
# ...
j main
```

35

Part a

Circle one answer (true or false) for each statement.

The keyboard input routine uses polling.		true	false
The keyboard input routine uses memory-mapped I/O.		true	false
The keyboard input routine uses interrupt-driven I/O.		true	false
The modem output routine uses polling.		true	false
The modem output routine uses memory-mapped I/O.		true	false
The modem output routine uses interrupt-driven I/O.		true	false

false, true, true, true, true, false

Part b

This program has a bug. If a keyboard interrupt occurs during a certain part of the main routine, the program will malfunction. Fill in the following two blanks, and briefly explain your answer.

The program will malfunction if the interrupt occurs after the execution of line ______ but before the execution of line ______.

Briefly explain your answer.

The problem occurs when the interrupt happens while the main program is dealing with the stack pointer, lines 22-28 (i.e. after 21, before 29). The interrupt handler increments the stack pointer and puts a character onto the stack, but then back in main (assuming the stack was not empty to begin with) the stack pointer is reset to its former state in line 28. The newly arrived character is thus in hyperspace with respect to the stack, and will be lost.

Part c

When the program malfunctions, it will (choose one) …

- fail to transfer one of the keystrokes to the modem.
- transfer one of the keystrokes to the modem twice.
- get stuck in an infinite loop.
- crash due to an invalid memory access.

Briefly explain your answer.

Choice a as just described.

Part d

True or false: Because the next key pressed is stored at address 0x00FF000A, the computer must have at least FF000A hex bytes of memory in order to operate correctly. Briefly explain your answer.

False. Memory-mapped i/o uses fake addresses. 2 points: 1 for answer, 1 for explanation.

You could also have earned the explanation point by stating that 0x00FF000A was a virtual address.

Problem 11 (pipelining)

Part a

Consider the C function below.

```
struct node {
      int data;
      struct node * next;
};
/* Return true if p points to a node whose next field is the same as p.
int is1nodeCircular (struct node * p) {
      return (p != null && p == p->next);
}
```

Provide an assembly language implementation of is1nodeCircular that

- returns the same result as the C version in all cases; and
- uses as few pipeline cycles as possible.

You may assume that forwarding, branch, and load delays are as described in Patterson and Hennessy section 6.1:

- the ALU output in one cycle may be used in an ALU operation in the next cycle,
- a register may be read and written in the same cycle,
- the instruction following a branch or jump is always executed,
- the result of a load cannot be used until the clock cycle after the load is in the MEM stage, and
- branches are resolved in stage 2.

```
Total 12 instructions

is1nodeCircular:
   beqz  $a0,false    # p == null if jump
   nop
   lw    $t0,4($a0)
   nop
   nop
   bne   $a0,$t0,false      # p != p->next if jump
true:
   move  $v0,$a0      # ok for branch delay
   jr    $ra
   nop
false:
   move  $v0,$0
   jr    $ra
   nop

Total 10 instructions

is1nodeCircular:
   beqz  $a0,done     # p == null if jump
   move  $v0,$a0      # $v0 == false
   lw    $t0,4($a0)
   nop
   nop
   beq   $t0,$a0,done
   nop
   move  $v0,$0
```

```
done:
   jr    $ra
   nop

Total 7 instructions

is1nodeCircular:
   beqz  $a0,done    # p == null if jump
   move  $v0,$0      # $v0 == false
   lw    $v0,4($a0)
   xor   $v0,$a0,$v0 # 0 iff equal
   sltiu $v0,$v0,1   # 1 in $v0 if equal, 0 if not
done:
   jr    $ra
   nop
```

Part b

How many pipeline cycles does your solution use if is1nodeCircular returns true? Draw a pipeline diagram or draw arrows indicating dependencies between instructions to explain your answer.